

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



UI视频教程及源代码+本书实例源代码

免费获取作者精心录制的“7天玩转iOS UI开发视频教程”，
总计36堂课，播放时长超过13小时



张益琿 编著

iOS开发实战

从零基础到App Store上架

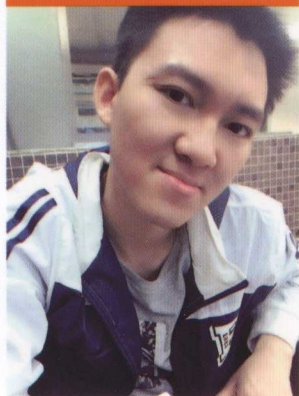
iOS 9+Xcode 7+Objective-C

开发兼容优雅的iOS应用，详解产品开发的全流程

清华大学出版社



作者介绍



张益珲

软件开发工程师，拥有多年iOS开发经验，曾开发iOS平台系列——游戏疯狂越狱1~2、应用物通配货软件、VIPExam考试库、证券财经软件等，现就职于中国唯品会。

iOS开发实战

从零基础到App Store上架

张益珩 编著

清华大学出版社

北京

内 容 简 介

《iOS 开发实战：从零基础到 App Store 上架》一书由一线软件工程师结合实际应用编写而成，由浅入深系统地介绍了 iOS 应用从开发、调试到打包、上架的完整过程。本书主体由各个基础模块组成，由实战项目连接，在帮助读者掌握原理的同时轻松上手开发出自己的应用。

为方便读者学习，作者还为本书精心录制了“7 天玩转 iOS UI 开发视频教程”，本视频教程包括基础篇、中级篇、高级篇、进阶篇、扩展篇 5 部分，总计 36 堂课，播放时长超过 13 小时。此外，本书还提供 iOS UI 开发视频教程源代码以及本书实例源代码。

本书的特色是通俗易懂，突出实战，提供了大量开发案例，适合于刚入职或新手 iOS 开发人员和爱好者、大中专院校学生及 iOS 培训班学员，尤其适合有一定语言基础想要开发 App 产品的开发者。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目 (CIP) 数据

iOS 开发实战：从零基础到 App Store 上架/张益琿编著. —北京：清华大学出版社，2016
ISBN 978-7-302-44184-7

I. ①i… II. ①张… III. ①移动终端—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆 CIP 数据核字 (2016) 第 148616 号

责任编辑：王金柱

封面设计：王 翔

责任校对：闫秀华

责任印制：李红英

出版发行：清华大学出版社

网 址：<http://www.tup.com.cn>, <http://www.wqbook.com>

地 址：北京清华大学学研大厦 A 座 邮 编：100084

社 总 机：010-62770175 邮 购：010-62786544

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

印 装 者：清华大学印刷厂

经 销：全国新华书店

开 本：190mm×260mm

印 张：27

字 数：691 千字

版 次：2016 年 8 月第 1 版

印 次：2016 年 8 月第 1 次印刷

印 数：1~3000

定 价：69.00 元

产品编号：068781-01

前言

编写本书的目的

近些年,移动端应用的开发越来越火热,市场对移动端两大主流操作系统平台 Android 与 iOS 开发者的需求量也大幅增加。很多大学毕业生或者相关行业从业者都有进入移动开发领域的想法,本书成书的原因,也正是为了解决这类群体人员的学习困扰。

开发一款完整的 iOS 软件是一个复杂的过程,开发者除了需要有编程语言的基础外,还需要对程序设计有宏观的把控。目前市面上大多数的教材,要么过于理论基础,看不到实在的学习效果,消磨读者兴趣;要么入门台阶过高,使读者无法顺利地进行学习。本书在编写时,定位的目标是帮助并无太多基础的读者快速上手 iOS 应用开发,通过一本书,完整地、了解 iOS 移动应用开发的整个过程,并且有能力自己开发一款常规的 iOS 移动应用。要做到这一点并不容易,在编写时除了要做到“事必躬亲”,不遗漏任何一个操作细节外,还要在讲解中插入完整的实战演示,以便读者能够学以致用,以用为学。

本书主要内容

本书分 11 章,下面介绍各章的主要内容与之间的联系。

第 1 章是为学习应用开发做准备,其中将介绍开发环境的搭建与开发工具的使用,这一章虽然为准备阶段,但对初学者来说却至关重要。

第 2 章将介绍 iOS 开发中的一些基础 UI 控件,移动端应用一个很重要的特点就是要要有绚丽的界面,应用程序的界面决定了用户使用这款应用程序的体验与心情,这一章向读者独立地介绍每个基础控件的用法,并通过实战提高读者综合使用这些控件的能力。

第 3 章在第 2 章的基础上,将向读者介绍 iOS 开发中经常使用到的更多高级控件的用法,同样也会为读者提供实战机会。

第 4 章主要讲解了 iOS 应用开发中的网络编程技术,由于网络编程的演示需要有网络数据支持,很多有关网络教学的文档书籍都只讲授理论,没有办法使读者切身地进行测试与练习。在编写本章时,特意注意了这个问题,本章除了讲授网络编程在 iOS 应用开发中的相关知识外,还将教读者如何使用网上免费的 API 服务真正做出一款网络应用。

第 5 章主要讲解 iOS 应用程序开发中的音频与视频技术,这类技术在开发音频软件和视频软件中意义重大。

第 6 章将作为动画专题向读者介绍 iOS 应用开发中的动画技术,章节设计由易到难,并且都配有代码演示。

第 7 章将作为传感器专题向读者介绍 iOS 开发中可以调用的设备传感器的相关知识。

第 8 章是界面布局专题,笔者参阅了很多 iOS 教材,其中都没有过多提到界面布局的相关知

识，这是一个十分大的弊端，界面布局技术是衡量一个开发者是否合格的重要指标，笔者相信读者学习 iOS 开发技术，绝对不只是想简简单单地做出一个 Demo 自己玩，做出“产品”才是读者的真正目标，而一款成熟的产品一定是具有兼容性的，一定是优雅的。因此，本书特别将 iOS 界面布局技术作为单独的一章来向读者介绍。

第 9 章是数据持久化专题，本章将介绍有关 iOS 应用开发中的文件操作及数据库操作的相关知识。

通过前 9 章的学习，读者能够具备独立开发一款 iOS 应用的基本能力，但是仅仅做出产品还不够，如何让自己的产品在市场发布，使用户可以下载使用也是开发者不得了解学习的内容，第 10 章将完整地向读者介绍提交自己的应用到 App Store 的整个过程。

第 11 章是进阶内容，此章也是读者开发能力提升的一章，本章将介绍一些独立于前面章节，但在实际开发中举足轻重的编程技术。

配书资源下载

为方便读者学习，作者还为本书精心录制了“7 天玩转 iOS UI 开发视频教程”，本视频教程包括基础篇、中级篇、高级篇、进阶篇、扩展篇 5 部分，总计 36 堂课，播放时长超过 13 小时。此外，本书还提供 iOS UI 开发视频教程源代码以及本书实例源代码。

读者可通过以下地址：<http://pan.baidu.com/s/1qXC2I0c>，获取本书 UI 视频教程及源代码。

读者也可以关注微信公众号 D11223344L 来获取 iOS 学习的相关资料或者加 QQ 群 203317592 与更多志同道合的朋友一起学习 iOS 开发技术。

如果读者遇到下载问题，请发电子邮件至 booksaga@126.com 联系，邮件主题为“求 iOS 开发实战配书资源”。

致谢

从接到出版社编辑的约稿邀请到本书初稿的完成，已经过去半年有余。本书中大部分内容都是在这半年间深夜的台灯下完成的。看着自己的作品，除了欣慰之外，更多地感觉到编程与代码已经成为了笔者生活中的一部分，除了工作与生计，从编程中得到的喜悦与满足感才是这项技能带给笔者的最大的礼物。同样，也希望读者热爱编程，从本书中除了学习到实用的技能外，还可以收获到更多乐趣。

在艺术的领域内，学无止境，编程也是一门艺术。笔者阅历尚浅，能力十分有限，在编写本书时，本力图将最完整的开发技巧与从事编程行业所积累的经验毫无保留地与读者分享，但在成书时，有些内容依然无法完整表达，笔者也相信，本书中可能会出现一些错误和遗漏，读者有任何疑问或建议，都可直接联系笔者本人。在生活中，笔者除了是一位全职开发者之外，也是一名编程教师，期待与读者互相帮助，共同进步。

最后，本书得以顺利完成，要感谢所有帮助过笔者的老师和朋友。感谢麦子学院 CEO 张凌华先生对笔者教学的鼓励和支持，感谢笔者的入门导师吕志轩老师，感谢支持笔者工作与写作的家人和笔者的女朋友，最重要的，感谢王金柱编辑在笔者写作期间一直给予的帮助和指导，没有他们，本书不可能来到读者的手中。

琿少

2016 年 5 月 25 日

目 录

第 1 章 开发准备	1
1.1 iOS 9 新特性简述	2
1.1.1 新增压力传感器编程接口	2
1.1.2 全新的搜索功能 API	2
1.1.3 更小、更快——全新的应用瘦身策略	3
1.1.4 使用更加安全的网络传输协议	4
1.2 熟悉 iOS 开发环境	4
1.2.1 安装 Xcode 开发工具	4
1.2.2 了解 Xcode 开发工具主界面	6
1.2.3 Xcode 开发工具的使用技巧及常用快捷键	7
1.3 创建第一个 iOS 项目	8
1.4 使用 Git 进行项目版本管理	13
1.4.1 Git 与 Github 简介	13
1.4.2 注册 GitHub 会员	13
1.4.3 使用 Xcode 创建 Git 仓库	14
1.4.4 用 Xcode 建立本地 Git 仓库与 GitHub 代码托管平台的联系	16
第 2 章 基础 UI 控件	19
2.1 iOS 系统 UI 框架的介绍	20
2.1.1 MVC 设计模式	20
2.1.2 代理设计模式	21

2.2 视图控制器——UIViewController	21
2.2.1 UIViewController 的生命周期	21
2.2.2 UIViewController 的视图层级结构	25
2.3 文本控件——UILabel	25
2.3.1 使用 UILabel 在屏幕上创建一个标签控件	26
2.3.2 自定义标签控件的相关属性	26
2.3.3 多行显示的 UILabel 与换行模式	27
2.4 按钮控件——UIButton	29
2.4.1 创建一个按钮来改变屏幕颜色	29
2.4.2 更加多彩的 UIButton 控件	32
2.5 文本输入框控件——UITextField	33
2.5.1 在屏幕上创建一个输入框	33
2.5.2 UITextField 的常用属性介绍	35
2.5.3 UITextField 的代理方法	36
2.5.4 实现一个监听输入信息的用户名输入框	37
2.6 开关控件——UISwitch	38
2.6.1 创建一个开关控件	38
2.6.2 为 UISwitch 控件添加触发方法	39
2.7 分页控制器——UIPageControl	40
2.8 分段控制器——UISegmentedControl	41
2.8.1 UISegmentedControl 基本属性的应用	41
2.8.2 对 UISegmentedControl 中的按钮进行增、删、改操作	42
2.8.3 UISegmentedControl 中按钮宽度的自适应	43
2.9 滑块控件——UISlider	43
2.9.1 UISlider 的创建与常规设置	44
2.9.2 对 UISlider 添加图片修饰	45
2.10 活动指示器控件——UIActivityIndicatorView	45
2.11 进度条控件——UIProgressView	47
2.12 步进控制器——UIStepper	48

2.12.1	步进控制器的基本属性使用	48
2.12.2	自定义 UIStepper 按钮图片	49
2.13	选择器控件——UIPickerView	49
2.13.1	创建一个 UIPickerView 控件	50
2.13.2	UIPickerView 选中数据时的回调代理	51
2.14	通过 CALayer 对视图进行修饰	52
2.14.1	创建圆角的控件	52
2.14.2	创建带边框的控件	52
2.14.3	为控件添加阴影效果	53
2.15	警告控制器——UIAlertController	54
2.15.1	UIAlertController 的警告框	54
2.15.2	UIAlertController 之活动列表	56
2.16	扩展篇	57
2.16.1	搜索栏控件——UISearchBar	57
2.16.2	日期时间选择器——UIDatePicker	59
2.16.3	警告视图——UIAlertView	61
2.16.4	活动列表——UIActionSheet	62
2.17	实战：登录注册界面的搭建	62
第 3 章	高级 UI 控件	68
3.1	导航控制器——UINavigationController	69
3.1.1	导航控制器的工作原理	69
3.1.2	使用导航控制器进行多界面搭建	70
3.1.3	导航栏 UINavigationController	73
3.1.4	导航按钮 UIBarButtonItem	74
3.1.5	导航控制器的工具栏	77
3.1.6	iOS 8 之后导航控制器的一些有趣功能	77
3.2	标签控制器——UITabBarController	78
3.2.1	标签控制器的工作原理	78
3.2.2	标签控制器的基础用法解析	78
3.2.3	关于 UITabBarItem 的使用	80

3.3 滚动视图——UIScrollView	81
3.3.1 使用 UIScrollView 展示视图内容	81
3.3.2 UIScrollView 的代理方法	83
3.4 网络视图——UIWebView	84
3.4.1 App 网络传输安全策略	85
3.4.2 通过网络请求加载 UIWebView	86
3.4.3 通过 HTML 字符串加载 UIWebView	86
3.4.4 通过 NSData 数据加载 UIWebView	87
3.4.5 UIWebView 中常用方法解析	88
3.4.6 UIWebView 的代理方法	89
3.5 表格视图——UITableView	90
3.5.1 UITableView 的创建与复用机制	90
3.5.2 创建一个表格视图 UITableView	91
3.5.3 关于表格数据的载体 UITableViewCell	93
3.5.4 设置 UITableView 的行高和头尾视图	95
3.5.5 UITableView 的用户交互行为	96
3.5.6 为 UITableView 添加索引栏	99
3.6 复杂布局视图——UICollectionView	99
3.6.1 UICollectionView 控件的优势与布局方式	100
3.6.2 使用 UICollectionView 进行九宫格式的布局	100
3.6.3 创建更加灵活的流式布局	102
3.6.4 自定义 UICollectionViewFlowLayout 进行参差瀑布流布局	103
3.6.5 使用 UICollectionView 进行圆环布局	106
3.7 实战：开发一款手机网页浏览器	109
3.7.1 网页浏览器工程的搭建	110
3.7.2 核心网页视图的设计	111
3.7.3 历史记录界面的设计	119
3.7.4 收藏界面的设计	122
3.7.5 启动页面、图标及应用名称的相关优化	124

第 4 章 网络编程	127
4.1 使用 NSURLConnection 请求网络数据	128
4.1.1 申请一个免费的 API 服务	128
4.1.2 使用 NSURLConnection 进行 API 服务数据的获取	131
4.1.3 使用 NSURLConnection 进行异步网络请求	132
4.1.4 使用 NSURLConnection 类通过代理回调的方式异步进行网络请求	134
4.2 设计封装一个更加易用的网络请求类	135
4.2.1 设计自定义的网络请求连接类	135
4.2.2 设计自定义的网络请求连接管理类	136
4.3 JSON 类型数据的解析与数据模型的设计	139
4.3.1 JSON 数据简介	139
4.3.2 在 iOS 中解析 JSON 数据	141
4.3.3 数据模型 Model 类的设计	142
4.4 使用 CocoaPods 进行第三方库的管理	146
4.4.1 在 MAC 上安装 CocoaPods	146
4.4.2 用 CocoaPods 搭建一个使用第三方网络请求框架 AFNetworking 的工程	148
4.5 使用 AFNetworking 进行网络请求	150
4.5.1 详解 HTTP/HTTPS 协议	150
4.5.2 使用 AFNetworking 进行网络请求	151
4.6 实战：开发“笑一笑”应用程序	153
4.6.1 工程项目框架的搭建	154
4.6.2 “笑一笑”界面数据载体 cell 的设计	155
4.6.3 “笑一笑”界面的搭建	157
4.6.4 实现下拉刷新与加载更多功能	162
4.6.5 “趣图吧”界面数据载体 cell 的设计	164
4.6.6 “趣图吧”界面的设计	167
第 5 章 音视频开发	172
5.1 iOS 音频开发基础——AVAudioPlayer 类的使用	173
5.1.1 使用 AVAudioPlayer 进行 MP3 音频文件的播放	173

5.1.2	进行音频播放相关属性的控制	175
5.1.3	后台播放音频及用户交互的优化	180
5.2	iOS 视频开发基础	184
5.2.1	使用 MPMoviePlayerController 向应用中嵌入视频模块	184
5.2.2	MPMoviePlayerController 常用属性与方法解析	185
5.3	视频播放器视图控制器——MPMoviePlayerViewController	189
5.4	AVPlayerViewController 视频播放框架与画中画开发技术	191
5.4.1	使用 AVPlayerViewController 进行视频播放	191
5.4.2	iPad 的画中画播放技术	193
5.5	实战：“天后王菲”音频播放器的开发	195
5.5.1	工程搭建与 LRC 歌词文件简介	196
5.5.2	LRC 歌词解析引擎的设计	197
5.5.3	核心播放器引擎的设计	201
5.5.4	歌曲列表与歌词显示视图界面的设计	208
5.5.5	播放器主页面的实现	213
5.5.6	后台播放音频用户交互的处理	219
第 6 章	动画开发	221
6.1	使用 UIImageView 播放图片组帧动画	222
6.2	UIView 层动画的应用	223
6.2.1	执行 UIView 层过渡动画的三个类方法	223
6.2.2	创建 UIView 层的阻尼动画	225
6.2.3	动画参数配置与组合动画	225
6.2.4	UIView 层过渡动画支持的属性	227
6.3	使用 commit 方式进行 UIView 层动画的创建	228
6.3.1	使用 commit 方式进行 UIView 层过渡动画的创建	228
6.3.2	两种 UIView 层动画创建方式的优劣	230
6.4	UIView 的转场动画	230
6.4.1	重绘 UIView 视图时使用的转场动画	230
6.4.2	切换 UIView 视图时使用的转场动画	231

6.5 核心动画编程技术——CoreAnimation	232
6.5.1 锚点对视图控件几何位置的影响	233
6.5.2 色彩梯度层——CAGradientLayer	234
6.5.3 视图拷贝层——CAReplicatorLayer	235
6.5.4 图形渲染层——CASHapeLayer	236
6.5.5 文本绘制层——CATextLayer	237
6.5.6 CAAAnimation 动画体系介绍	238
6.5.7 使用 CABasicAnimation 创建基础动画	240
6.5.8 使用 CAKeyframeAnimation 类创建关键帧动画	242
6.5.9 CALayer 层的转场动画——CATransition	243
6.5.10 CALayer 层的组合动画——CAAnimationGroup	245
6.5.11 CATransform3D 变换的应用	246
6.6 炫酷的粒子效果	248
6.6.1 粒子发射器——CAEmitterLayer	248
6.6.2 粒子单元——CAEmitterCell	250
6.6.3 创建粒子火焰动画	251
6.7 播放 GIF 动态图	253
6.7.1 使用 UIWebView 进行 GIF 动态图播放	253
6.7.2 使用 UIImageView 帧动画进行 GIF 动态图播放	254
6.8 实战：小游戏 Flappy Bird 的设计与开发	256
6.8.1 小鸟对象的设计	257
6.8.2 游戏开始界面的设计	259
6.8.3 游戏结束界面的设计	261
6.8.4 Flappy Bird 游戏主框架的搭建	262
第7章 传感器开发	270
7.1 为应用程序添加手机密码及指纹识别的安全验证	271
7.1.1 使用手机密码为应用程序添加安全验证	271
7.1.2 使用用户指纹为应用程序添加安全验证	273
7.2 使用加速度传感器、螺旋仪传感器与磁力传感器获取设备空间状态	274
7.2.1 使用 UIAccelerometer 获取设备空间状态	274

7.2.2 使用 CoreMotion 框架获取设备空间状态信息	275
7.3 距离传感器的应用	278
7.4 iOS 蓝牙开发技术	279
7.4.1 中心设备管理类 CBCentralManager	280
7.4.2 外围设备管理类 CBPeripheralManager	285
7.5 GPS 应用与地图编程技术	289
7.5.1 进行设备地理位置定位	289
7.5.2 原生地图开发技术	292
7.5.3 在地图中添加大头针及标注	294
7.5.4 在地图视图中添加覆盖物	297
7.5.5 在地图中进行线路导航与附近兴趣点检索	299
7.6 实战：简易蓝牙对战五子棋	304
7.6.1 游戏核心通信类的设计	304
7.6.2 棋盘瓦片的设计	314
7.6.3 核心游戏视图与游戏核心逻辑的设计	315
7.6.4 核心游戏视图控制器的设计	325
第 8 章 界面布局	329
8.1 iOS 中传统的 UIViewAutoresizing 布局模式	330
8.1.1 通过代码来设置视图控件的 UIViewAutoresizing 模式	330
8.1.2 在 xib 文件中可视化地配置控件的 autoresizing 属性	332
8.2 Autolayout 自动布局框架	333
8.2.1 初识 Autolayout	334
8.2.2 Autolayout 的属性意义与一个简单的自动布局示例	335
8.2.3 使用 Objective-C 风格的方法进行代码 Autolayout 布局	338
8.2.4 使用格式化的字符串进行 Autolayout 布局对象的创建	341
8.2.5 与约束相关的几个方法	343
8.2.6 使用 Autolayout 设计一个高度自适应的聊天输入框及动画优化	343
8.2.7 使用第三方库 Masonry 进行 Autolayout 约束布局	345

第 9 章 数据持久化	351
9.1 使用 plist 文件进行轻量级数据持久化管理	352
9.1.1 在工程中读取 plist 文件数据	352
9.1.2 在程序沙盒 Documents 目录中创建和使用 plist 文件	353
9.1.3 使用 UserDefaults 类进行数据持久化	354
9.2 使用归档技术进行数据模型持久化	356
9.2.1 进行单一系统数据类型的归档与解归档操作	356
9.2.2 对多个对象进行数据归档	357
9.2.3 进行自定义数据模型的归档	358
9.3 小型数据库 SQLite 在 iOS 开发中的应用	360
9.3.1 SQLite 数据库常用语法介绍	360
9.3.2 使用 iOS 原生框架 sqlite3 对 SQLite 数据库进行操作	362
9.4 核心数据管理框架 CoreData 的使用	367
9.4.1 使用 CoreData 设计数据模型	367
9.4.2 CoreData 编程框架中 3 个重要的类	370
9.4.3 CoreData 编程框架的数据操作	373
9.4.4 使用 CoreData 进行数据与页面的绑定	378
9.5 网络缓存策略	384
9.5.1 为网络请求设置缓存策略	384
9.5.2 应用缓存管理类 NSURLCache 简介	385
第 10 章 提交应用程序到 AppStore	387
10.1 使用 Xcode 开发工具进行程序调试	388
10.1.1 使用自定义断点进行代码调试	388
10.1.2 添加全局异常断点	389
10.1.3 使用 LLDB 调试器进行程序调试	390
10.2 Apple 开发者账号的申请	391
10.2.1 几种类型的开发者账号	391
10.2.2 申请开发者账号的过程	391

10.3 进行应用程序的打包	394
10.3.1 在 iTunes Connect 中进行应用的创建与配置	394
10.3.2 使用 Xcode 进行打包与提交 iTunes	401
第 11 章 进阶技巧	405
11.1 Objective-C 中 block 语法的应用	406
11.1.1 声明与实现 block 语法块	406
11.1.2 block 代码块中访问对象的微妙关系	407
11.2 iOS 通知中心 NSNotificationCenter 的应用	408
11.2.1 通知类 NSNotification 简介	409
11.2.2 通知中心 NSNotificationCenter 应用	409
11.3 多线程开发技术	410
11.3.1 使用 NSThread 进行线程管理	411
11.3.2 使用 NSOperation 类与 NSOperationQueue 类进行多任务管理	412
11.3.3 iOS 中 GCD 编程技术简介	416

第 1 章

开发准备

工欲善其事，必先利其器。在学习 iOS 移动开发之前，首先应该将开发环境配置完成并且对所需要使用的开发工具进行了解与熟悉。本章将首先向读者介绍 iOS 9 系统相比之前系统的一些新特性，使读者对目前主流的 iOS 系统有一宏观上的了解。后面将一步步演示开发环境的搭建并向读者介绍开发工具 Xcode 的常用功能。

通过本章的学习，读者能够掌握：

1. 了解 iOS 9 新特性和新功能。
2. 申请免费的 Apple ID 账号。
3. 使用 Xcode 开发工具创建 iOS 工程。
4. 使用 Xcode 开发工具编写与调试程序。
5. 熟悉 Xcode 工程结构。
6. 编写第一个程序 HelloWorld。
7. 使用 Git 工具进行版本管理。
8. 使用 GitHub 代码托管平台。

1.1 iOS 9 新特性简述

iOS 9 是 iOS 系统的一次重大升级，增强了 iOS 设备的安全性并支持新增传感器的硬件编程接口。多任务处理能力也更加优秀强大。熟悉 iOS 9 的这些改变支持，可以更好地帮助读者学习 iOS 应用程序的开发。

1.1.1 新增压力传感器编程接口

在 iPhone 6s 问世之后，很多果粉都争先要体验 3D Touch 给用户带来的额外维度上的交互，这个设计之所以叫作 3D Touch，其原理上是增加了一个压力的感触，通过区分轻按和重按来进行不同的用户交互。Xcode7 开始支持 3D Touch 的开发，遗憾的是，模拟器并不能进行 3D Touch 的操作调试，开发者需要在真机上进行测试调试。

3D Touch 功能在用户维度上主要有两个方面的应用，第一部分的应用是用户可以通过 3D 手势，在主屏幕上的应用图标处，直接进入应用的某些功能模块。这个功能就例如用户可以重按系统日历 App，会在日历 App 图标旁边出现一个菜单，单击菜单我们可以进入相应的功能单元。第二部分的应用是对 App 的一个优化，用户可以通过 3D Touch 手势在 View 视图上来预览一些预加载信息，这样的设计可以使 App 更加简洁大方，交互性也更强。如图 1-1 所示。

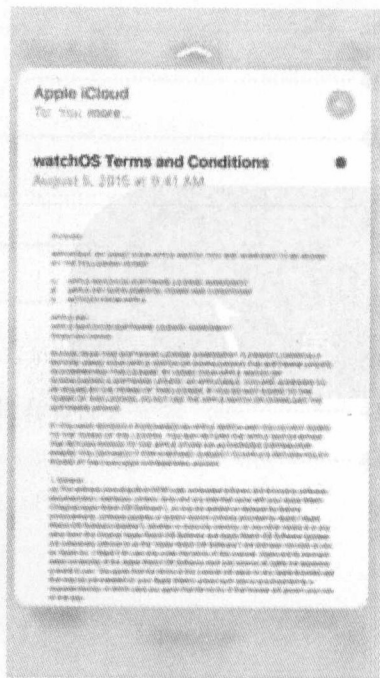


图 1-1 3D Touch 的预加载视图

1.1.2 全新的搜索功能 API

iOS 9 为开发者提供了许多新的编程 API，搜索功能的加强无疑是其中比较显眼的。首先，可以设想一下：如果在 App 中定义一种标识符，在 Siri 和搜索中，用户可以用这个标识符搜索到相应 App，是不是很棒？再扩展一些，如果可以定义任意的数据，使其在搜索和 Siri 中可以快速检索到。更进一步，开发者甚至可以在开发者的网站中根据协议添加一些标志，使 Apple 的爬虫可以检索到，那样，即使用户没有安装 App，也可以在搜索中获取到相应的信息，这是多么酷的事。

iOS 9 提供了 3 种全新的搜索模式。

第 1 种搜索模式是使用 `NSUserActivity` 活跃元素类，开发者可以使用它来为应用程序添加活跃元素，也就是说，用户可以在搜索中根据活跃元素搜索到开发者的 App。示例代码如下：

```

//创建一个对象,这里的 type 用于区分搜索的类型
NSUserActivity *userActivity = [[NSUserActivity alloc] initWithActivityType:@"myapp"];
//显示的标题
userActivity.title = @"我的 app";
// 搜索的关键字
userActivity.keywords = [NSSet setWithArray: @[@"sea",@"rch"]];
// 支持 Search
userActivity.eligibleForSearch = YES;
//提交设置
[userActivity becomeCurrent];

```

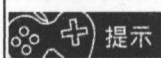
当用户通过相应的活跃元素找到 App, 并且激活 App 时, 系统会调用如下方法, 开发者可以在其中做相应的逻辑操作。

```

- (BOOL)application:(UIApplication *)application continueUserActivity:(NSUserActivity *)userActivity restorationHandler:
{
    NSString *activityType = userActivity.activityType;
    if ([activityType isEqual:@"myapp"]){
        // Handle restoration for values provided in userInfo
        // do something

        return YES;
    }
    return NO;
}

```



这里的代码只是作为示例, 读者可以不必深究其原理与使用方法, 等读者学习完后面的章节, 具备开发 iOS 实际应用的能力后, 可以再来学习此段代码。

第2种搜索模式是采用 CoreSpotlight 框架, CoreSpotlight 是一种更加自由的搜索方式, 可以通过添加一些模型, 将 App 中的数据展示在搜索栏中, CoreSpotlight 框架类似提供了一些增、删、改、查的操作, 开发者可以自由地进行搜索属性的设置。

第3种搜索模式为 Web Markup 模式, 这个功能与 App 端开发关系不大, 但是对 App 的推广却至关重要, 这项技术可以让 App 关联一个网站, Apple 通过爬虫来获取我们规定的一些标签值, 无论用户是否安装了此 App, 在搜索时, 都可以展示出相关信息。

1.1.3 更小、更快——全新的应用瘦身策略

Apple 在 iOS 9 中引入了一套新的 App 瘦身方案, 通过一些优化策略, 尽可能地减小 App 安装包的体积。这部分的大多数工作是由 AppStore 来完成的, 开发者并不需要付出太多额外的开销。

App 的瘦身策略主要包括 3 部分。

第 1 部分为 Slicing，这部分的主要原理是在不同的设备中下载安装不同的 App 包，各个 App 包副本中包含相应尺寸的素材。在 Xcode 中，开发者使用 Asset Catalog 管理素材文件，在提交应用市场后，AppStore 系统会自动帮开发者生成各个尺寸包的 App 副本。

第 2 部分是使用 BitCode 编码，字节码文件是 App 程序的一种中间形式，Apple 会对包含字节码的 App 进行二次优化，来进行相应的瘦身。

第 3 部分是 On-Demand Resources，这是一种多级应用的设计思路，例如一个游戏，开发者可以将其分为一个个大小各异的资源包，用户只需下载一个小的引导程序，在程序内加载相应的资源包。这样就可以大大加快应用的下载安装速度。

1.1.4 使用更加安全的网络传输协议

随着 iOS 9 的推出和 Xcode 的升级，Apple 将默认开发者使用 Https 的传输方式，相比 Http 的传输协议，这会增加一些安全性，对于开发者而言，一下子将 Http 协议全部升级为 Https 协议不是一件容易的事，但是开发者可以通过 Xcode 的一些配置，使其支持 Http 的传输协议，这将是一种折中的方案。

具体 Xcode 的配置方式，在后面章节使用到时会向读者演示过程。

1.2 熟悉 iOS 开发环境

进行 iOS 应用开发，必备的开发软件便是 Xcode，Xcode 开发工具十分强大并且简单易用，不需要过多的配置，下载安装后，各种环境和模拟器就关联安装好了，对于初学时使用来说门槛很低。

1.2.1 安装 Xcode 开发工具

和 Android 有些不同，开发 iOS 的平台并不多，Xcode 是 Apple 公司自己开发的一套针对 OS、iOS、watchOS 和 tvOS 的开发环境，使用方便并且功能十分强大。可以在 AppStore 上免费获取 Xcode 开发工具。

1. 首先需要申请个人的 AppID

AppID 是 Apple 会员的凭证，也是个人的信息管理凭证。申请个人的 AppID 也是免费的，登录 www.apple.com/cn 苹果（中国），在屏幕右上角的购物袋按钮中选择登录选项，如图 1-2 所示。

在登录界面右下侧选择创建一个 Apple ID，如图 1-3 所示。

之后按照网页的指示填写相应信息，需要注意的是，填写的邮箱务必要真实，注册 Apple ID 时会要求进行邮箱验证。

Apple ID 申请成功之后，就可以从 AppStore 获取 Xcode 开发工具了。打开 AppStore，如图 1-4 所示。



图 1-2 苹果官网



图 1-3 登录界面

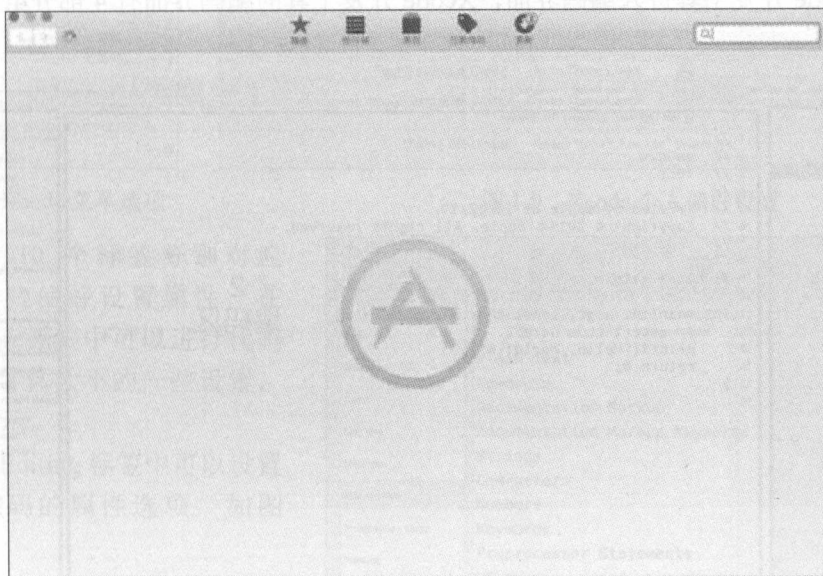


图 1-4 AppStore 应用市场

在右上角的搜索框中输入 Xcode, 按 return 键进行搜索, 会搜索出许多应用, 其中第一个就是开发者需要的 Xcode 开发工具, 单击获取安装到我们的电脑即可, 如图 1-5 所示。

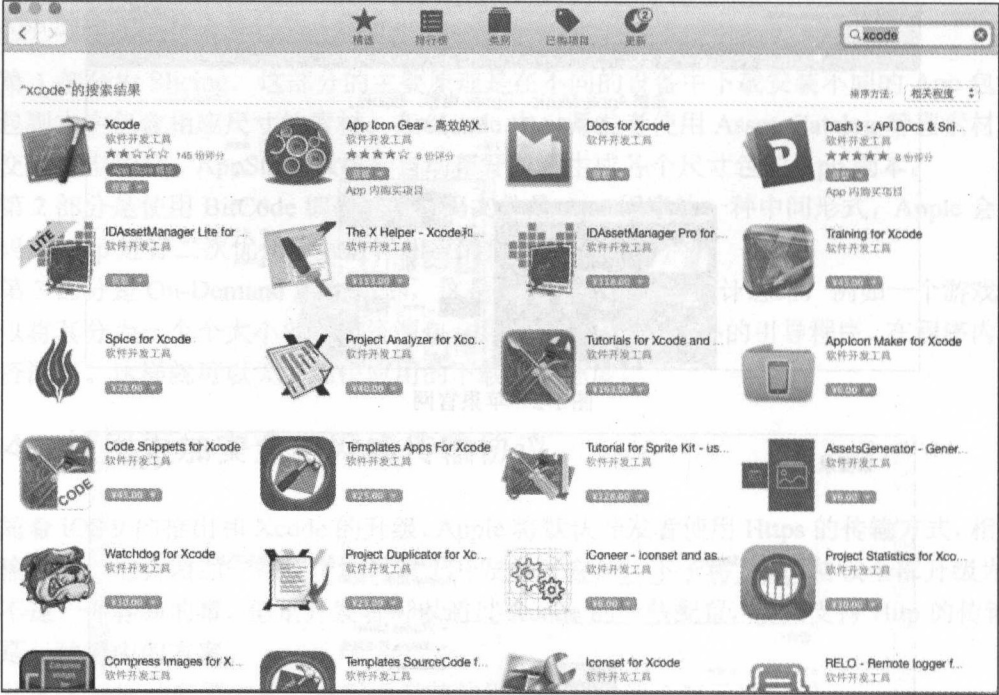


图 1-5 获取 Xcode 开发工具

1.2.2 了解 Xcode 开发工具主界面

打开 Xcode 开发工具进入编码界面，Xcode 开发工具的编码界面有 4 部分组成，如图 1-6 所示。

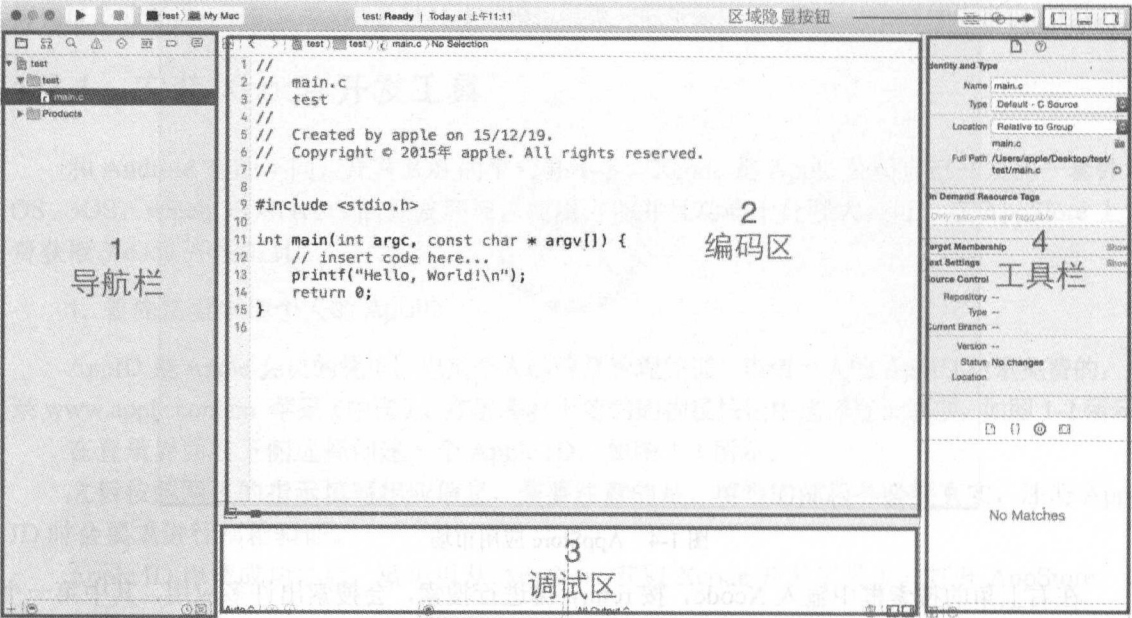


图 1-6 Xcode 开发工具的编码界面

图 1-6 中最左边是导航栏，其中展示一些类似文件目录索引、关键字搜索索引、错误警告索引、断点调试索引等。中间区域是编码的主要区域，在这个区域中编写相关的程序代码，下边的区域是 debug 调试区域，代码中的打印信息会展示在这个区域中，最右边是工具栏，用于设置当前编写文件的相关属性。界面的右上角有 3 个按钮，从左向右分别对应了导航栏、调试区和工具栏的显隐，在编码时，可以将暂时不需要使用到的区域隐藏，扩大编码区域。

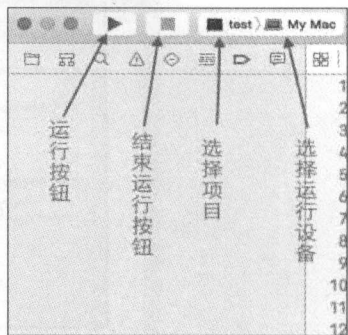


图 1-7 程序调试相关功能

在 Xcode 主界面的左上角也有一些按钮，如图 1-7 所示。

其中运行按钮可以编译并且运行项目，选择项目按钮可以选择需要运行的项目，选择运行设备按钮可以对运行的平台进行选择。

1.2.3 Xcode 开发工具的使用技巧及常用快捷键

熟练地使用 Xcode 可以使开发变得事半功倍，Xcode 也有许多附加功能可帮助开发者更高效地进行代码编写。单击 Xcode 标签导航中的 preferences 选项，如图 1-8 所示，之后会弹出如图 1-9 所示的个人偏好窗口。

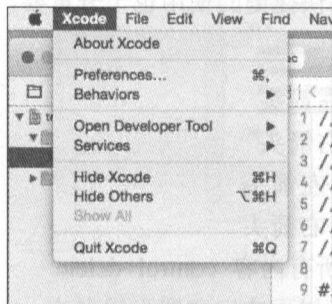


图 1-8 单击 Xcode 菜单选项

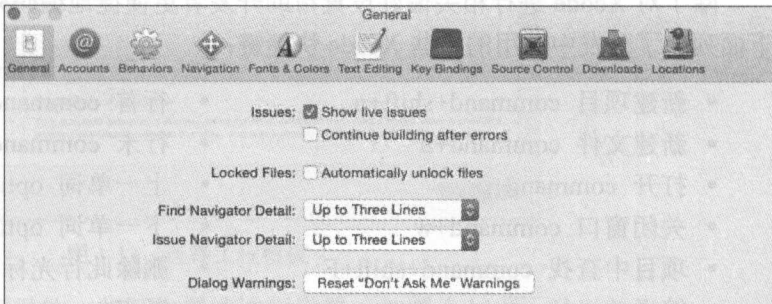


图 1-9 Xcode 个人偏好设置

上面的 10 个标签分别对应 Xcode 的一些偏好设置属性，在 Fonts&Colors 标签中可以进行代码高亮风格和字体大小的一些设置，如图 1-10 所示。

在 Text Editing 标签中可以设置一些编辑代码的属性选项，如图 1-11 所示。

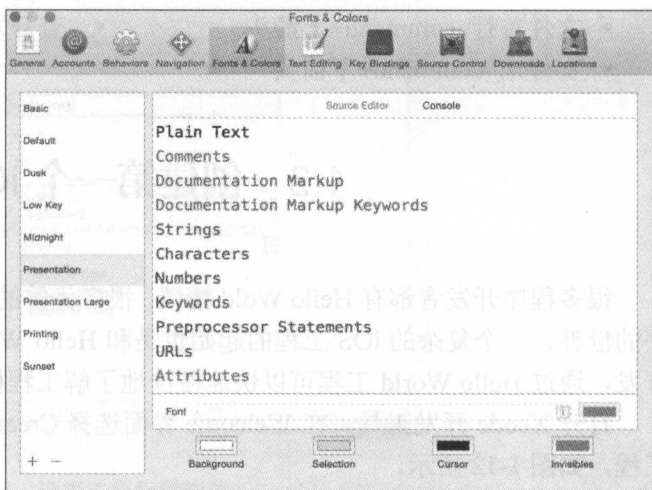


图 1-10 风格字体设置界面

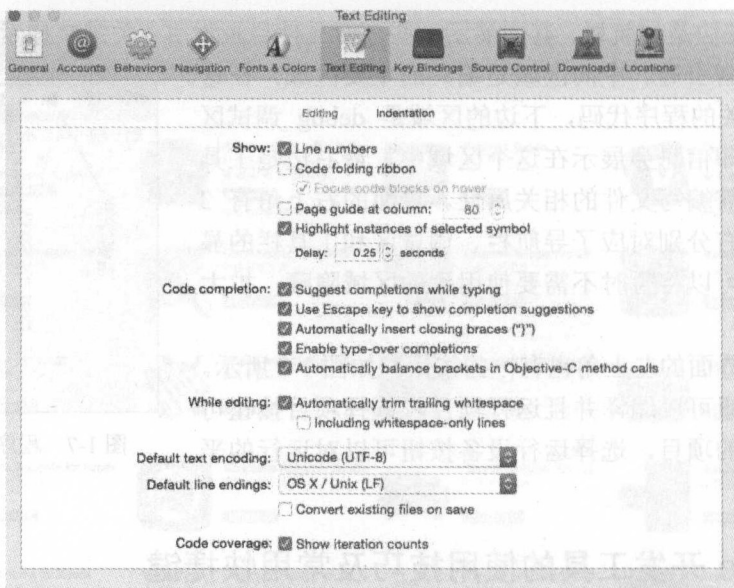


图 1-11 代码编辑选项

其中 Line numbers 可以设置是否显示代码行号，在编写代码时，最好选中这一项，它对开发者调试代码定位问题十分重要。

除了对 Xcode 进行相关偏好设置帮助开发者更便捷地编码外，快捷键的使用也十分重要，下面列出了开发中常用的一些 Xcode 快捷键：

- 新建项目 `command+shift+n`
- 新建文件 `command+n`
- 打开 `command+o`
- 关闭窗口 `command+w`
- 项目中查找 `command+shift+F`
- 编译并运行 `command+r`
- 注释 `command+/`
- 文件首行 `command+上箭头`
- 文件末 `command+下箭头`
- 行首 `command+左箭头`
- 行末 `command+右箭头`
- 上一单词 `option+左箭头`
- 下一单词 `option+右箭头`
- 删除此行光标前所有内容 `control+delete`
- 断点 `command+option+b`
- 当前行插入断点 `command+\`
- 查开发文档 `command+option+click`

1.3 创建第一个 iOS 项目

很多程序开发者都有 Hello World 情愫，很多优秀的开发者也是通过 Hello World 进入了程序的世界。一个复杂的 iOS 工程的起始也是和 Hello World 工程有相同的结构，因此学习 iOS 开发，通过 Hello World 工程可以快速便捷地了解工程框架。

打开 Xcode 开发工具，在 Welcome 界面选择 Create a new Xcode project 选项来新建一个工程，如图 1-12 所示。

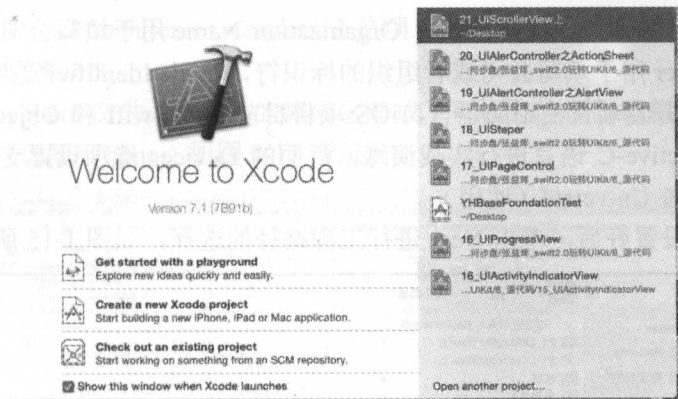


图 1-12 使用 Xcode 创建一个新的工程

在选择模板窗口中选择 Single View Application，如图 1-13 所示。

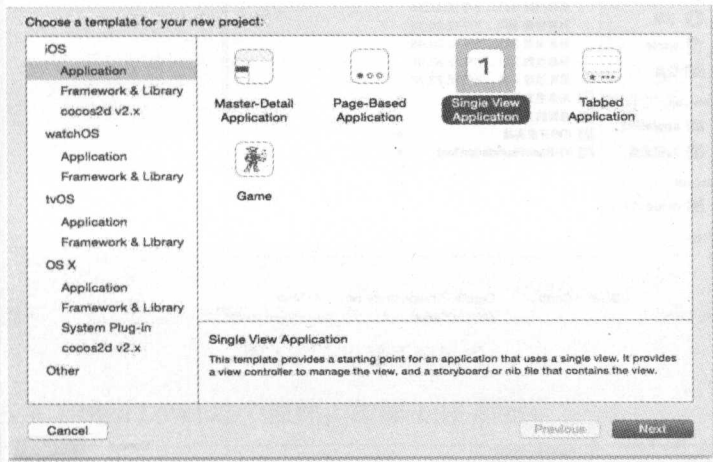


图 1-13 选择工程模板

在模板设置窗口中可以对项目的一些基本属性进行设置，如图 1-14 所示。

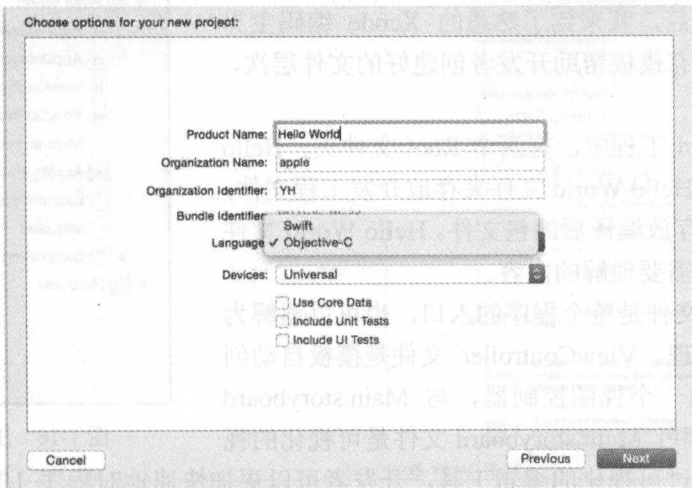


图 1-14 设置工程配置选项

Product Name 用于设置项目的名称, Organization Name 用于填写公司或者组织的名称, Organization Identifier 用于填写公司或者组织的标识符, Bundle Identifier 是当前项目的标识符, Language 可以选择开发项目使用的语言, iOS 项目目前支持 Swift 和 Objective-C 两种语言, 本书主要采用 Objective-C 语言进行实战演练, 后面的 Devices 选项设置支持的设备, 可以选择 iPhone、iPad 或者 Universal (通用)。

将上面的信息设置好后, 单击 Next 进行工程路径的选择, 如图 1-15 所示。

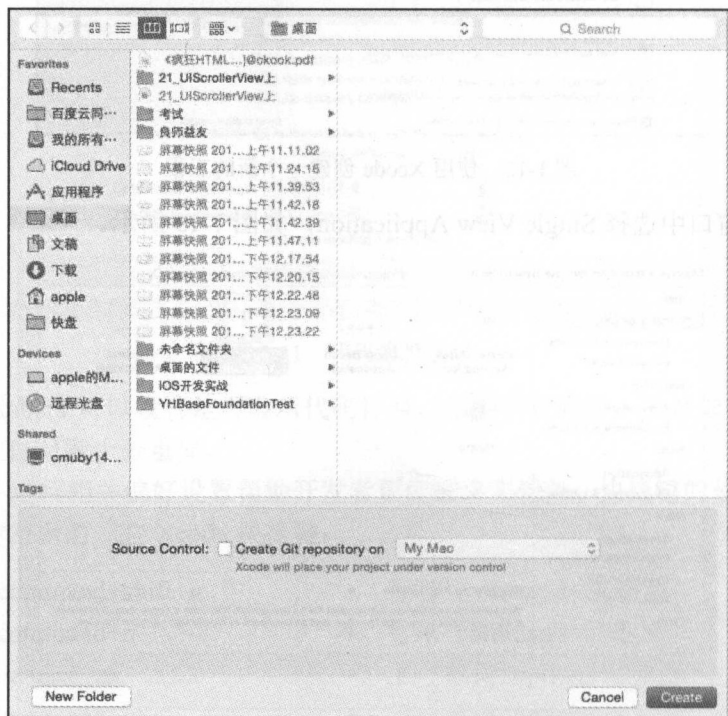


图 1-15 选择工程存储位置

这里将其保存在桌面, 单击 Create 进行工程的创建。

工程创建完成后, 就来到了熟悉的 Xcode 编码主界面, 左侧导航栏中有模板帮助开发者创建好的文件层次, 如图 1-16 所示。

在 Hello World 工程中, 有两个 Root 文件夹, Hello World 和 Products。Hello World 文件夹存放开发工程文件, Products 文件夹中存放编译后的包文件。Hello World 文件夹中的文件是重点需要理解的内容。

AppDelegate 文件是整个程序的入口, 也可以理解为 iOS 程序运行的代理。ViewController 文件是模板自动创建出展示在屏幕的一个视图控制器, 与 Main.storyboard 中的视图控制器关联; Main.storyboard 文件是可视化的视图编辑器文件, 通过可视化的编辑工具, 开发者可以更加快速地对程序 UI 部分进行开发; Assets.xcassets 文件是图片素材文件管理器, 如果项目中需要使用到一些图片素材, 可以将图

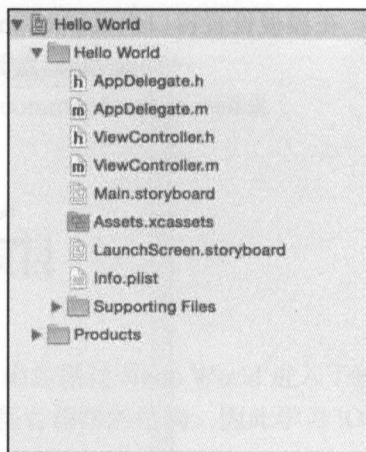


图 1-16 工程目录结构

片放入这个管理器中；LaunchScreen.storyboard 是项目初启画面的视图管理器，Info.plist 文件保存了项目的一些配置属性。

打开 Main.storyboard 文件，Xcode 的编码区变成了可视化的视图编辑区，取消选中的 use size classes，使其只适配 iPhone，如图 1-17 所示。

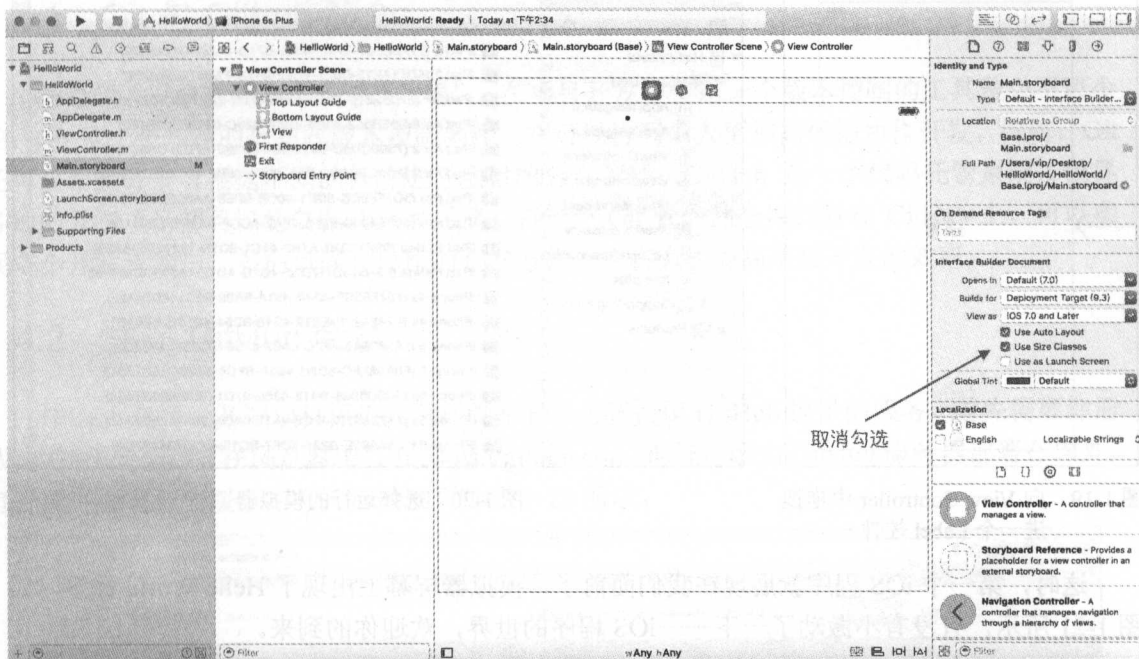


图 1-17 设置适配模式

在编辑器的右下方找到 Label 这个控件，如图 1-18 所示。

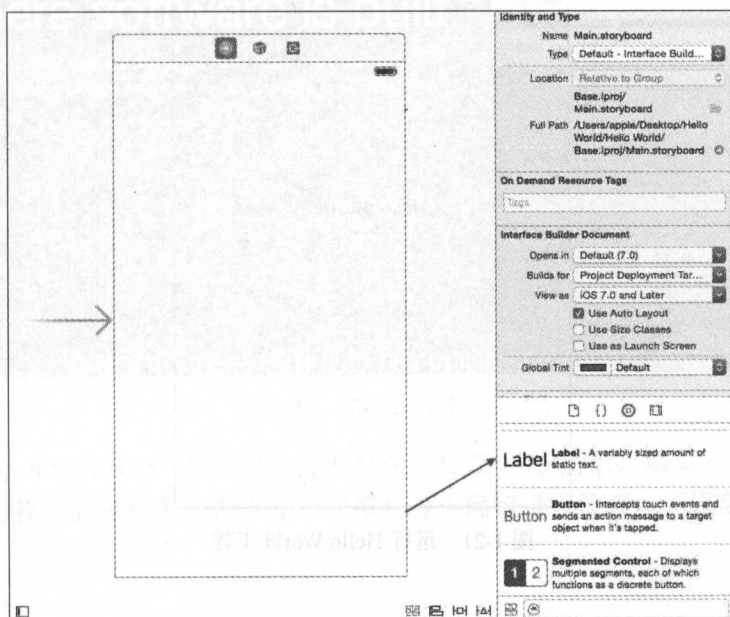


图 1-18 storyboard 文件中的 Label 控件

使用鼠标点住 Label 不放，将其拖动到视图控制器的中间，如图 1-19 所示。

双击视图控制器上的 Label，在其中写入 Hello World 字样，之后单击 Xcode 左上角的运行按钮，选择一个模拟器，如图 1-20 所示。

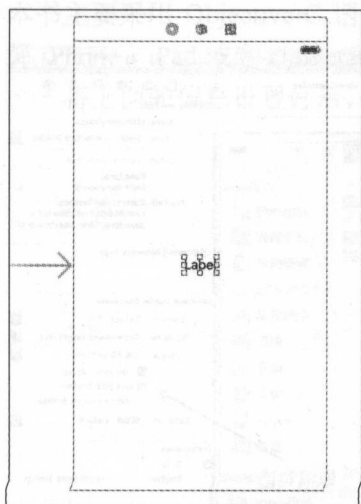


图 1-19 向 View Controller 中拖拽
进入一个 Label 控件



图 1-20 选择运行的模拟器

这时，第一个 iOS 程序就展现在我们面前了，模拟器屏幕上出现了 Hello World 标签，如图 1-21 所示，有没有小激动了一下——iOS 程序的世界，欢迎你的到来。

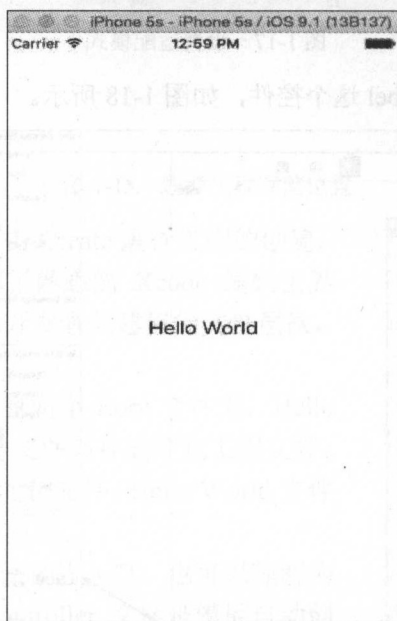


图 1-21 运行 Hello World 工程

1.4 使用 Git 进行项目版本管理

1.4.1 Git 与 Github 简介

俗话说“工欲善其事，必先利其器”。在项目开发中使用一个版本控制的工具是必不可少的，Git 是一个开源的分布式版本控制系统。它可以协同多人更加高效地协作开发，同时，Git 还可以帮助开发者根据不同的目的进行项目的分支管理。GitHub 是一个代码托管系统，世界各地的开源项目都可以免费在其上面托管。将一个 Git 管理的仓库托管在 GitHub 上，可以实现多个开发者参与，多地点同时协作的开发方式，这将大大提高项目开发的效率。

1.4.2 注册 GitHub 会员

GitHub 免费为开源项目提供代码托管平台，若要使用 GitHub 提供的服务，首先需要注册成为 GitHub 会员。在浏览器中打开 <https://github.com> 地址。因为 GitHub 服务器部署在国外，打开或许会有一些缓慢。打开后的网页如图 1-22 所示。

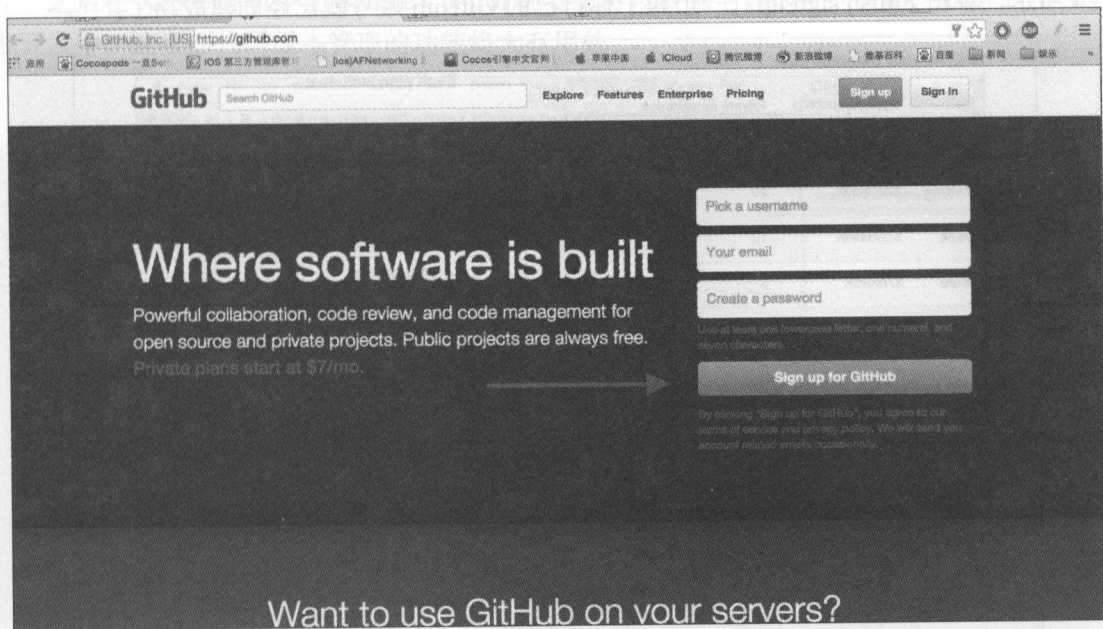


图 1-22 GitHub 主页

单击“Sign up for GitHub”按钮进入注册界面，如图 1-23 所示，填写一些基本信息，这之间，github 会对我们的用户名是否重复，邮箱是否正确等进行检查，无误后单击“creat an account”按钮。

The best way to design, build, and ship software.

Step 1: Set up a personal account Step 2: Choose your plan Step 3: Go to your dashboard

Create your personal account

There were problems creating your account.

Username
jakiZhang ✓

Email Address
783147203@qq.com ✓
You will occasionally receive account related emails. We promise not to share your email with anyone.

Password
***** ✓

By clicking on "Create an account" below, you are agreeing to the Terms of Service and the Privacy Policy.

Create an account

You'll love GitHub

Unlimited collaborators
Unlimited public repositories

- ✓ Great communication
- ✓ Friction-less development
- ✓ Open source community

图 1-23 填写注册信息

如果注册成功，GitHub 会让我们选择服务类型，作为个人开发者，可以选择 free，如图 1-24 所示，单击 Finish sign up，一个属于你自己的 GitHub 账号就已经创建成功了。

Choose your personal plan

Plan	Cost (view in CNY)	Private repositories	
Large	\$50/month	50	Choose
Medium	\$22/month	20	Choose
Small	\$12/month	10	Choose
Micro	\$7/month	5	Choose
Free	\$0/month	0	Chosen

Charges to your account will be made in US Dollars. Converted prices are provided as a convenience and are only an estimate based on current exchange rates. Local prices will change as the exchange rate fluctuates.
Don't worry, you can cancel or upgrade at any time.

☐ Help me set up an organization next
Organizations are separate from personal accounts and are best suited for businesses who need to manage permissions for many employees.
[Learn more about organizations.](#)

Finish sign up

Each plan includes:

Unlimited collaborators
Unlimited public repositories

- ✓ Free setup
- ✓ HTTPS Protection
- ✓ Email support
- ✓ Wikis, Issues, Pages, & more

图 1-24 选择服务类型

1.4.3 使用 Xcode 创建 Git 仓库

Xcode 是一个系统一体化性很强的 iOS 开发工具，在安装 Xcode 时默认也安装了 Git 系统，在我们创建工程的时候可以选择创建本地 Git 仓库，如图 1-25 所示。

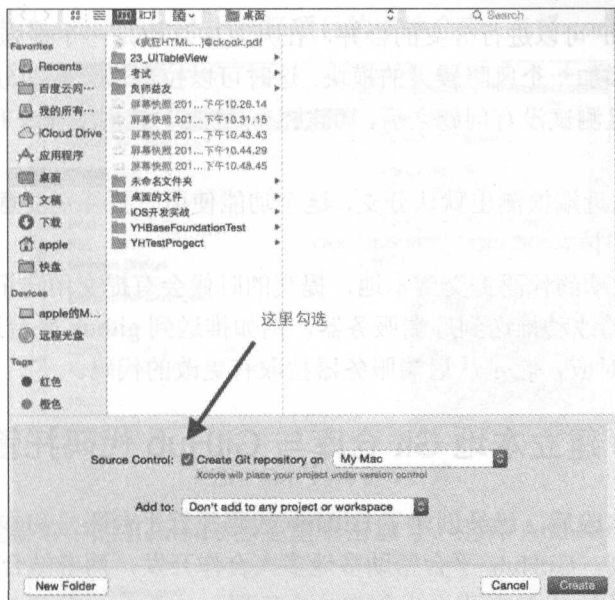


图 1-25 创建 Git 仓库

之后在 Xcode 导航中的 Source Control 标签里可以看到当前项目的仓库，如图 1-26 所示，在开发中，这里面代码版本管理的功能将大有用处。

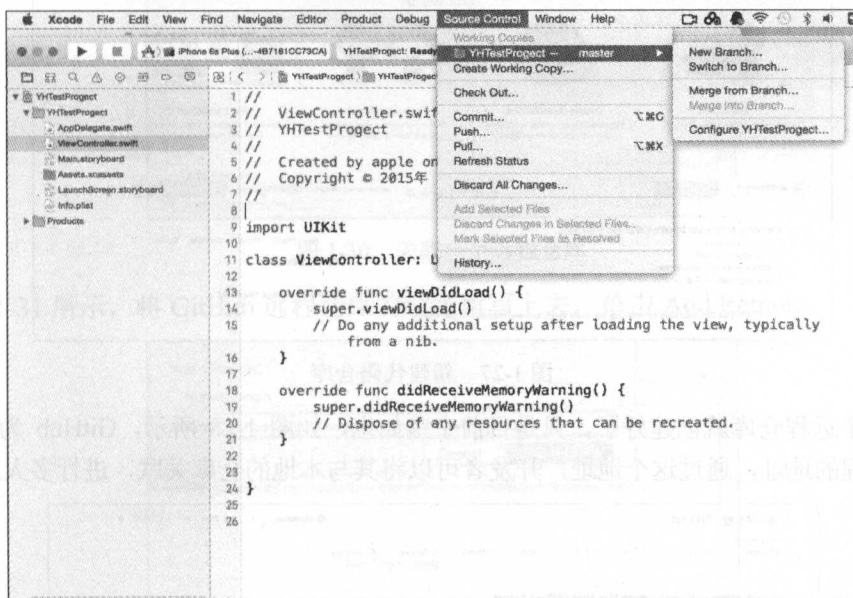


图 1-26 Xcode 的代码管理功能

New Branch 可以创建一个新的分支，分支创建时是一个副本，但是其可以在不影响其他分支的情况下独立开发新的扩展。举个例子，项目的初始版本是 1.0，现在需要开发 2.0 的版本，开发者就完全可以在 1.0 版本的基础上拉出一个 2.0 的分支，在 2.0 分支上做的开发工作都不会影响 1.0 版本。

Switch to Branch 提供切换分支功能，开发者可以在多个分支之间自行安排，灵活开发。

Merge from Branch 可以进行分支的合并，在开发中这也是一个很强大的功能。例如开发者需要在当前工程中添加一个风险较大的模块，这时可以拉出一个新的分支，在新的分支上进行开发，开发完成并且测试没有问题之后，可以在原分支上使用 Merge from Branch 进行代码合并。

Check Out 可以从远端检测出默认分支，这个功能使用时要特别注意，如果本地分支中文件有改动，将会被覆盖掉。

Commit 可以将改动的代码提交到本地，提交的时候会有提交用户记录和备注操作。

Push 功能将本地的改动推送到远端服务器，例如推送到 github 平台进行托管。

Pull 功能与 Push 对应，它是从远端服务器拉取有更改的代码。

1.4.4 用 Xcode 建立本地 Git 仓库与 GitHub 代码托管平台的联系

上面的一些步骤完成后，已经创建了 GitHub 代码托管平台账号和本地 git 仓库，git 仓库版本进行本地版本控制，GitHub 平台帮助多地多人合作开发，两者结合才能最高效地进行项目的开发。首先，需要在 GitHub 平台上创建一个远程 repository（仓库），用申请好的账号登录 GitHub 进入主页，如图 1-27 所示，单击 New repository，需要注意的是，如果是新创建的 GitHub 账号，需要先进行邮箱验证。

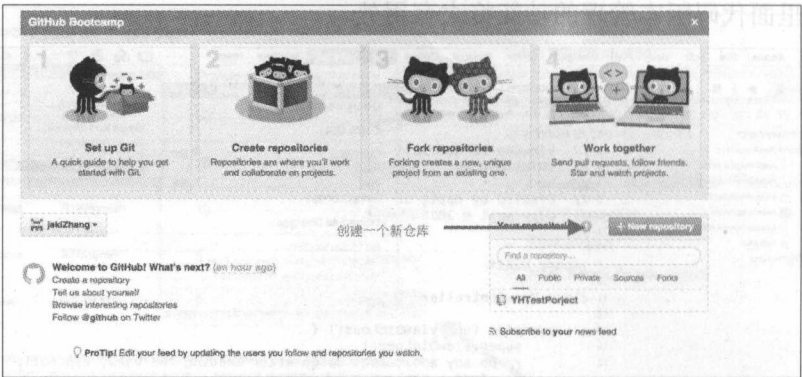


图 1-27 新建代码仓库

这样一个远程仓库就创建好了，只是目前空空如也，如图 1-28 所示，GitHub 为这个仓库分配了一个远程的地址，通过这个地址，开发者可以将其与本地的仓库关联，进行多人远程协作。

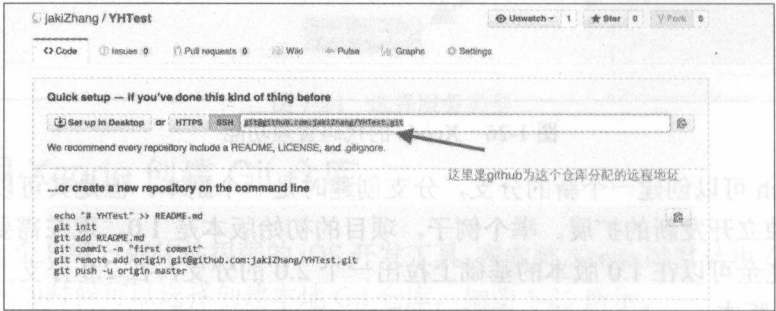


图 1-28 仓库地址

再回到 Xcode, 在 Source Control 中选择项目的本地仓库, 选择 configure, 如图 1-29 所示。

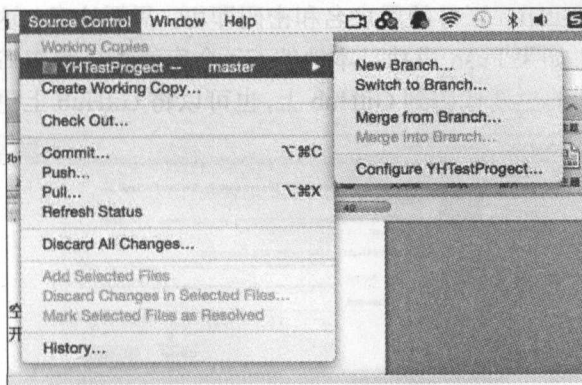


图 1-29 进行代码仓库设置

在弹出来的设置菜单中的 Remotes 标签里单击加号, 选择 Add Remote, 如图 1-30 所示。

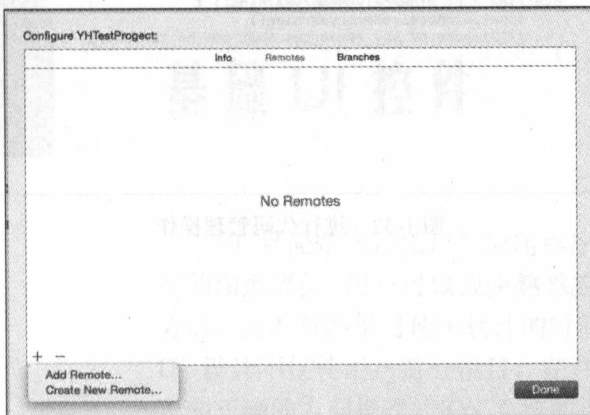


图 1-30 关联一个远程仓库

如图 1-31 所示, 将 GitHub 远程仓库的地址填写上去, 单击 Add Remote。

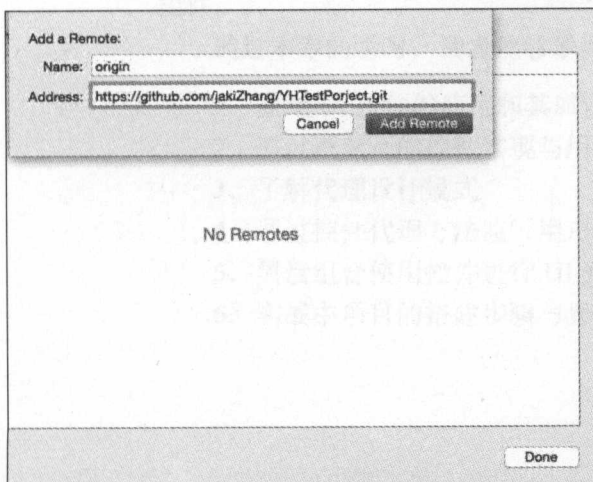


图 1-31 关联远程从库地址

然后使用 Push 功能，将本地的代码 Push 到 GitHub 上，第一次使用时 Push 会需要我们输入用户名与密码，使用 GitHub 账号的用户名和密码即可，需要注意，这里的用户名不是邮箱，是 GitHub 会员用户名。如果 Push 成功，本地的 Git 仓库就和托管在 GitHub 上的仓库进行了关联，我们可以随时随地的更新代码到 GitHub 上，也可以将 GitHub 上更新的代码拉到本地来。

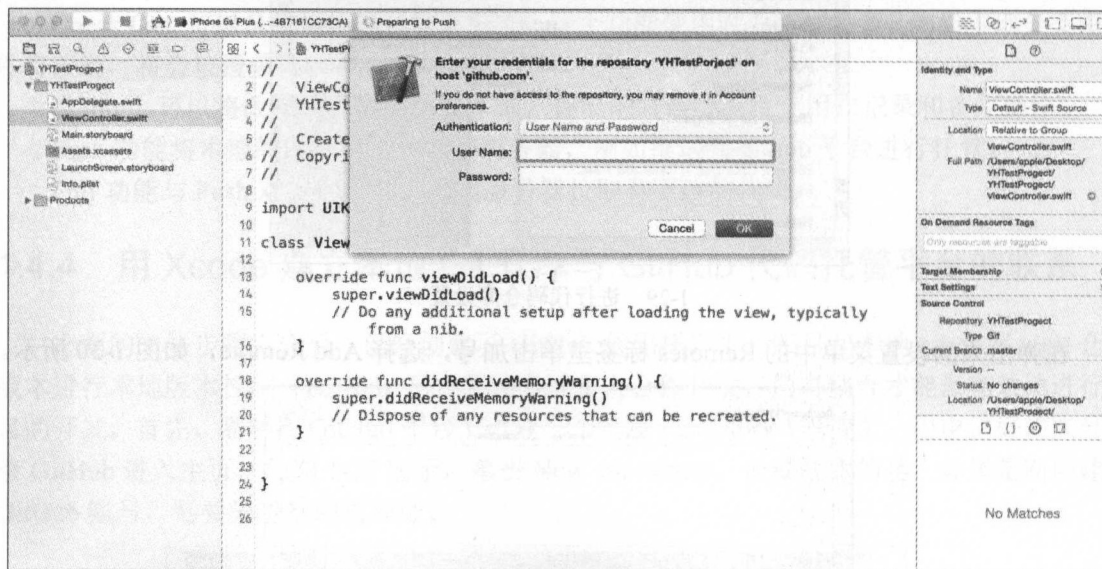


图 1-32 进行代码管理操作

第 2 章

基础 UI 控件

UI 界面是连接用户与应用程序的桥梁，通过美观易用的图形界面，用户可以很快熟悉应用程序的功能和操作方法，大大节约学习使用软件的时间成本。同时，成功的 UI 设计不仅使用户赏心悦目，在许多情况下也会引导用户向正确的方向操作。iOS 系统的 UI 框架十分简约美观且易于开发者进行自定义，本章将向读者介绍讲解 iOS 系统的 UI 体系架构及 iOS 应用开发中常用的一些独立 UI 控件。

通过本章的学习，读者能够掌握：

1. 基础 UI 控件的使用和其属性的自定义
2. 通过可交互的控件实现与用户操作进行逻辑交互
3. 了解代理设计模式
4. 通过控件代理方法监听用户操作
5. 灵活组合使用控件进行 UI 开发
6. 实战中项目的搭建步骤与界面间的跳转

2.1 iOS 系统 UI 框架的介绍

在 iOS 系统的开发框架中，UIKit 是专门负责界面渲染的一个框架，其中不仅封装了许多开发中常用的 UI 控件，如 UILabel（标签）控件，UIButton（按钮）控件等，UIKit 还集成了建立视图联系的视图控制器和交互用户操作的用户触摸事件等。UIKit 框架的结构十分复杂，学习 UIKit 框架时，首先需要从大局着眼，熟悉框架的构成与其中模块间的联系，之后再从细节入手，从常用的控件类入手学起，多多积累，时常练习，由局部到整体逐渐掌握 iOS 开发中 UI 的相关部分。

如图 2-1 所示为 UIKit 框架中类的结构与简易关系。UIKit 框架中大致分为 3 个部分：

- ① 独立的 UI 视图控件。
- ② 充当视图控件载体的控制器类。
- ③ 管理触摸事件、手势操作、键盘操作等交互的管理类。

独立于 UIKit 框架之外，代理和设计模式的设计模式为开发者提供了 UI 属性变化的回调接口。由于各个模块是独立的，开发者可以十分方便地进行 UI 上的自定义与扩展操作，同时各个模块又各有接口关联，其在数据传递和逻辑交互上也井井有条毫不混乱。

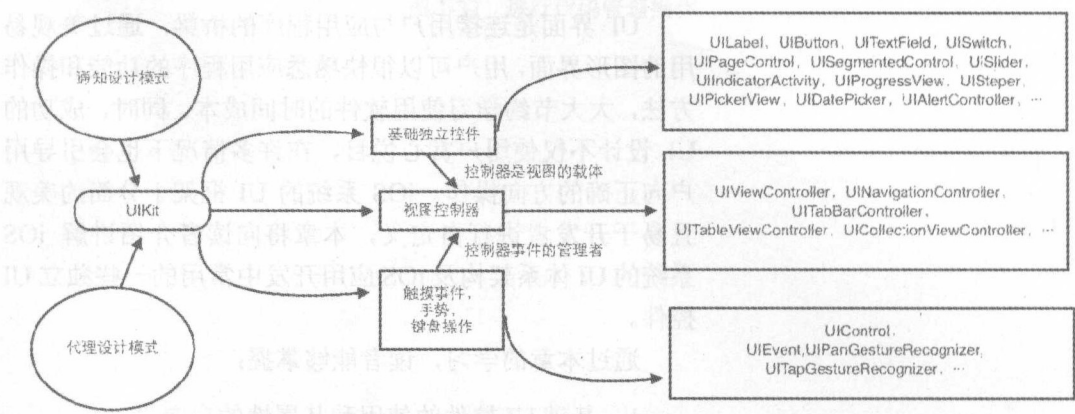


图 2-1 UIKit 框架的类结构与简易关系图

2.1.1 MVC 设计模式

MVC 模式是一种传统的软件设计模式，其全称为 Model（模型）-View（视图）-Controller（控制器）。在这种模式中，重新设计 UI 界面时不需要重复编写数据和逻辑相关的代码。在 MVC 模式中，数据由数据模型 Model 层管理，UI 视图由 View 层管理，将 View 与数据进行关联，处理逻辑的相关操作交由控制器 Controller 层进行管理，View 层与 Controller 层的分离，使得许多 View 可以进行复用，在一些开源的开发者社区也可以看到大量自定义的 View 层的控件，在项目中需要时，几乎不需要修改就可以直接使用。Model 层与 View 层的分离使得

Model 层不再依赖 View 层,在数据模型没有改变的情况下,View 层可以方便地重新进行设计。

在 UIKit 框架中,基础的独立控件就充当 View 层,控制器部分充当 Controller 层,数据层一般由开发者自行进行模型的设计和编写,在 Controller 中通过控件的功能接口将数据与控件关联。

2.1.2 代理设计模式

代理设计模式是软件编程中进行数据传递的一种重要方式,这种模式十分抽象但在 UIKit 框架中却随处可见。举一个例子来说明,古时候,盲人背着一个瘸子走路,瘸子看到前面有一个水坑,他告诉盲人让他绕过去,盲人听从瘸子的指挥,两人平安到达目的地。在这一组行为中,瘸子看到前面有障碍,可是瘸子不能走路;另一方面,盲人可以走路,但是他看不到,他不知道这个障碍是否存在。于是,语言沟通就充当了瘸子与盲人之间的代理,瘸子看到障碍后通过语言沟通将这个信息告诉盲人,盲人获取信息,绕过障碍。类比编程开发中,系统 UI 控件充当着故事中的瘸子的角色,开发者充当着盲人的角色,当系统 UI 控件接收到用户的一些动作行为时它并不知道应该怎么处理这些行为逻辑,于是通过代理方法将用户的这一动作告诉开发者,开发者在代理方法中处理相关的程序逻辑。



提示

代理这种设计模式比较抽象,对于初学者可能很难理解,不过读者不用担心,在后面具体讲解控件时会使用代理方法来处理交互逻辑,到时候可以再结合本小节内容进行理解,就会容易很多。

2.2 视图控制器——UIViewController

UIViewController 是 UIKit 框架中 Controller 部分的基础,一些复杂的 Controller 类也是基于 UIViewController 继承出来的。在开发应用程序时,所有界面也都是基于 UIViewController 搭建出来的。

2.2.1 UIViewController 的生命周期

生命周期是指一个对象从创建出来到其被释放销毁的整个过程。Objective-C 和 Swift 都是面向对象的高级语言,为了保持内存的平衡与程序运行的高效,当需要一个对象时,它会被创建并给它分配内存空间;同样,当它不再被需要时,也应该被系统释放回收。在一个 UIViewController 对象从创建到释放过程中,会依次调用许多生命周期函数,了解这些函数的调用时机和功能是 iOS 开发者的必修课。打开 Xcode 开发工具,创建一个名为 UIViewControllerTest 的工程,将使用的开发语言选择为 Objective-C。工程创建出来后,系统的模板自动生成一个 ViewController 类,这个类是继承于 UIViewController 并且与 Main.storyboard 中的初始视图控制器关联。简单来说,这个类创建了一个视图控制器作为工程的根视图控制器,我们可以在这个类中编写相关代码来对 UIViewController 的生命周期进行研究。

UIViewController 中与生命周期相关的函数有很多,列举如下:


```

//类的初始化方法
+ (void)initialize;
//对象初始化方法
- (instancetype)init;
//从归档初始化
- (instancetype)initWithCoder:(NSCoder *)coder;
//从 nib 文件初始化
- (void)awakeFromNib;
//加载视图
- (void)loadView;
//将要加载视图
- (void)viewDidLoad;
//将要布局子视图
- (void)viewWillLayoutSubviews;
//已经布局子视图
- (void)viewDidLayoutSubviews;
//内存警告
- (void)didReceiveMemoryWarning;
//已经展示
- (void)viewDidAppear:(BOOL)animated;
//将要展示
- (void)viewWillAppear:(BOOL)animated;
//将要消失
- (void)viewWillDisappear:(BOOL)animated;
//已经消失
- (void)viewDidDisappear:(BOOL)animated;
- (void)didReceiveMemoryWarning;
//被释放
- (void)dealloc;

```

实践是最好的老师，在编程的学习中亦是如此，要了解上面方法的执行次序，跟踪程序的运行是最快的方法，先在 ViewController.m 文件中将这些方法实现，代码如下：

```

#import "ViewController.h"
int tip=0;
@interface ViewController ()

@end

@implementation ViewController
+ (void)initialize{
    [super initialize];
    NSLog(@"%d initialize", ++tip);
}
- (instancetype)init

```

```
{
    self = [super init];
    if (self) {

    }
    NSLog(@"%d init", ++tip);
    return self;
}
- (instancetype)initWithCoder: (NSCoder *)coder
{
    self = [super initWithCoder:coder];
    if (self) {

    }
    NSLog(@"%d initWithCoder", ++tip);
    return self;
}
- (void)awakeFromNib{
    [super awakeFromNib];
    NSLog(@"%d awakeFromNib", ++tip);
}
- (void)loadView{
    [super loadView];
    NSLog(@"%d loadView", ++tip);
}
- (void)viewDidLoad {
    [super viewDidLoad];
    NSLog(@"%d viewDidLoad", ++tip);
}
- (void)viewWillLayoutSubviews{
    [super viewWillLayoutSubviews];
    NSLog(@"%d viewWillLayoutSubviews", ++tip);
}
- (void)viewDidLayoutSubviews{
    [super viewDidLayoutSubviews];
    NSLog(@"%d viewDidLayoutSubviews", ++tip);
}
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    NSLog(@"%d didReceiveMemoryWarning", ++tip);
}
- (void)viewWillAppear: (BOOL)animated{
```

```

    [super viewWillAppear:animated];
    NSLog(@"%d viewWillAppear", ++tip);
}

-(void)viewDidAppear:(BOOL)animated{
    [super viewDidAppear:animated];
    NSLog(@"%d viewDidAppear", ++tip);
}

-(void)viewWillDisappear:(BOOL)animated{
    [super viewWillDisappear:animated];
    NSLog(@"%d viewWillDisappear", ++tip);
}

-(void)viewDidDisappear:(BOOL)animated{
    [super viewDidDisappear:animated];
    NSLog(@"%d viewDidDisappear", ++tip);
}

-(void)dealloc{
    NSLog(@"%d dealloc", ++tip);
}

@end

```

上面代码中，创建一个全局变量 `tip`，使用这个变量来对程序的运行过程进行标记。`ViewController` 是 `UIViewController` 的一个子类，在子类中覆写父类的方法时，需要先调用父类的此方法。关于 `self` 和 `super` 这两个关键字一直是初学者的噩梦，甚至在许多时候，拥有一定开发经验的开发者也不能完全理解其意义。在这里读者只需要记住，在实例方法中（又称减方法，以减号开头），`self` 指调用这个方法的对象，即当前类的一个实例对象，如果用 `super` 调用方法则代表从父类中找这个方法实现，而在类方法中（又称加方法，以加号开头），`self` 指当前类，如果用 `super` 调用方法则代码从父类中找当前类方法的实现。



提示

`dealloc` 方法是唯一一个不需要并且也不能在实现里调用父类方法的函数，这个函数在 ARC(自动引用计数)环境中不再被开发者所需要，但是开发者依然可以重写这个函数来监测内存的释放情况。

运行工程，在 Xcode 的调试区会打印出如图 2-2 的信息。

```

2016-01-12 22:46:55.118 UIViewControllerTest[676:32564] 1 initialize
2016-01-12 22:46:55.120 UIViewControllerTest[676:32564] 2 initWithCoder
2016-01-12 22:46:55.120 UIViewControllerTest[676:32564] 3 awakeFromNib
2016-01-12 22:46:55.126 UIViewControllerTest[676:32564] 4 loadView
2016-01-12 22:46:55.126 UIViewControllerTest[676:32564] 5 viewDidLoad
2016-01-12 22:46:55.126 UIViewControllerTest[676:32564] 6 viewWillAppear
2016-01-12 22:46:55.131 UIViewControllerTest[676:32564] 7 viewWillLayoutSubviews
2016-01-12 22:46:55.131 UIViewControllerTest[676:32564] 8 viewDidLayoutSubviews
2016-01-12 22:46:55.133 UIViewControllerTest[676:32564] 9 viewDidAppear

```

图 2-2 跟踪程序运行的打印信息

从打印信息中可以清晰地看到一个 `UINavigationController` 被创建的过程中依次会调用的方法，但是

这些信息并不全面，并没有体现出 `UIViewController` 销毁时的过程，也没有展现从不同来源初始化的 `UIViewController` 生命周期的不同。在列出的生命周期方法中，`initialize` 方法比较特殊，这个函数并不会在每次创建对象时都调用，只在这个类第一次创建对象的时候会调用做一些类的准备工作，实际上，如果有继承的子类，如果子类没有实现这个 `initialize` 方法，当第一次创建子类对象时父类会代替子类再调用一次 `initialize` 方法。

`init` 和 `initWithCoder` 方法作用相似，都是对对象做初始化工作，如果从代码进行初始化则会调用 `init` 方法，从归档文件进行初始化则会调用 `initWithCoder` 方法。`awakeFromNib` 方法会在从 `xib` 或者 `storyboard` 中加载的 `UIViewController` 将要激活时被调用。

`loadView` 方法是开始加载 UI 视图的初始方法，这个方法除非开发者手动调用，否则在 `UIViewController` 的生命周期中只会被调用一次。

`viewDidLoad` 方法在视图已经加载完成后会被调用，因为这个函数调用的时候，Controller 的基本系统功能已经初始化完成，开发者一般会将一些 Controller 额外定义功能的初始化工作放在这个函数中。

`viewWillAppear` 方法在视图即将显示的时候调用。

`viewWillLayoutSubviews` 方法在视图将要布局其子视图时被调用。

`viewDidLayoutSubviews` 方法在视图布局完成其子视图时被调用。

`viewDidAppear` 方法在视图已经显示后被调用。

上面这些方法是 `UIViewController` 从创建到展现出来之间调用的生命周期函数，这些只是一半，`UIViewController` 被释放和销毁的过程由如下方法完成：

- `viewWillDisappear` 方法在视图将要消失时调用，开发者可以在其中做一些数据清理的操作。
- `viewDidDisappear` 方法在视图已经消失时被调用。
- `dealloc` 方法是对象的销毁方法，在对象被释放时调用，开发者可以通过在其中打印信息的方式检查一个类是否存在内存泄露等问题。

2.2.2 `UIViewController` 的视图层级结构

`UIViewController` 自带一个 `UIView` 类型的 `view`，这个 `view` 平铺在屏幕上，是 Controller 的根视图，如果在 Controller 中添加其他 UI 控件都是添加在这个 `view` 上，`UIView` 类通过 `addSubview` 方法来添加子 `view` 视图，子 `view` 视图也可以继续使用 `addSubview` 方法添加它自己的子视图，在第 1 章中，Hello World 标签实际上就是添加在 Controller 的根 `view` 上的一个子 `Label` 视图，iOS 系统通过这样的层级结构管理整个 UI 体系。

2.3 文本控件——`UILabel`

在任意一款应用中，随处可见各种各样的文字标签，`Label`（标签）是应用 UI 体系中最基础也是最简单的一个控件。顾名思义，其主要作用是在屏幕视图上显示一行或者多行的文本，

十分类似于生活中的便条标签。在 UIKit 框架中，UILabel 有很多可定制属性提供给开发者进行控件的自定义设置。

2.3.1 使用 UILabel 在屏幕上创建一个标签控件

在第一章中，通过 storyboard 文件很轻松的在视图上创建了显示 Hello World 的标签，这一小节我们使用代码来实现同样的效果。打开 Xcode 开发工具，创建一个名为 HelloWorldText 的工程，在 ViewController.m 中的 viewDidLoad 方法里添加如下代码：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UILabel * label = [[UILabel alloc] initWithFrame:CGRectMake(20, 100, 280, 30)];
    label.text = @"Hello World";
    [self.view addSubview:label];
}
```

上面代码中，initWithFrame: 方法是 UILabel 类中的一个初始化方法，这个方法中需要传入一个 CGRect 类型的结构体，这个参数决定了在屏幕上创建 UILabel 控件的位置和尺寸，CGRect 在 iOS 的 UI 系统中描绘了一个矩形区域，CGRectMake() 方法可以构造出一个 CGRect 结构体，其中的 4 个参数依次设置了这个矩形区域的 x 坐标、y 坐标、宽度和高度。



提示

在 UIKit 框架中，UI 坐标系与生活中的数学坐标系略有不同，生活中横向为 x 轴，向右增大，纵向为 y 轴，向上增大，在 UI 坐标系中，横向为 x 轴，向右增大，纵向为 y 轴，向下增大，即原点在左上角。

UILabel 类的 text 属性用于设置标签上的文字，必须设置为一个 NSString 类型的字符串值。在调用 UIView 的 addSubview: 方法后，将 UILabel 控件添加在了当前的视图上，运行工程会看到如图 2-3 的效果。

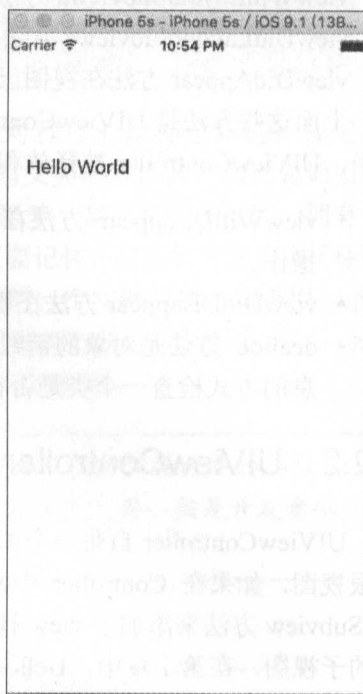


图 2-3 代码创建的 UILabel 控件

2.3.2 自定义标签控件的相关属性

上面创建的 HelloWorldText 工程中标签控件的样式是系统默认的模样，有时候，开发者需要更多元化的标签，比如各种颜色，各种字体等，UILabel 中有大量的属性提供给开发者进行自定义，代码示例如下：

```

- (void)viewDidLoad {
    [super viewDidLoad];
    UILabel * label = [[UILabel alloc] initWithFrame:CGRectMake(20, 100, 280, 3
0)];
    label.text = @"Hello World";
    //设置背景颜色
    label.backgroundColor = [UIColor redColor];
    //设置字体和字号
    label.font = [UIFont systemFontOfSize:23];
    //设置字体颜色
    label.textColor = [UIColor whiteColor];
    //设置对齐模式
    label.textAlignment = NSTextAlignmentCenter;
    //设置阴影颜色
    label.shadowColor = [UIColor greenColor];
    //设置阴影的偏移量
    label.shadowOffset = CGSizeMake(10, 10);
    [self.view addSubview:label];
}

```

`backgroundColor` 属性设置了标签的背景颜色，实际上 `backgroundColor` 并非是 `UILabel` 特有的属性，很多通过继承 `UIView` 来的子类都有这样一个属性。`font` 属性设置 `UILabel` 控件上的字体相关属性；`textColor` 属性设置了 `UILabel` 控件上字体的颜色；`TextAlignment` 属性设置 `UILabel` 控件上文字的对齐模式，默认是居中对齐，设置对齐模式使用如下枚举值：

```

typedef NS_ENUM(NSInteger, NSTextAlignment) {
    NSTextAlignmentLeft      = 0,    // 居左对齐
    NSTextAlignmentCenter    = 1,    // 居中对齐
    NSTextAlignmentRight     = 2,    // 居右对齐
};

```

`shadowColor` 属性设置文字的阴影颜色，`shadowOffset` 属性设置了阴影的偏移量，即阴影与本体之间的偏移距离，这个属性要设置一个 `CGSize` 类型的结构体，`CGSize` 中的两个参数分别代表横向偏移量和纵向偏移量。通过添加上面的代码，再次运行工程，效果如图 2-4 所示。



图 2-4 自定义属性的 UILabel

2.3.3 多行显示的 UILabel 与换行模式

通过 `initWithFrame:` 初始化方法创建的 `UILabel` 控件，会有一个宽度，如果文字长度超过了 `UILabel` 控件的宽度，默认的 `UILabel` 控件并不会换行，而是用省略号代替超出的部分，例如将 `UILabel` 控件高度和 `text` 属性改为如下所示：

```
UILabel * label = [[UILabel alloc] initWithFrame: CGRectMake(20, 100, 280, 100)];
label.text = @"Hello World,It is a good idea, So,what do you want to konw?";
```



提示

UILabel 可以多行显示的前提是有足够的高度。

运行工程，如图 2-5 所示，会看到多出的文字被截断了，UILabel 并没有换行。
默认的 UILabel 都是单行显示的，可以通过如下属性设置显示的行数：

```
label.numberOfLines = 0;
```

`numberOfLines` 设置为一个整数值，代表支持多少行显示，如果设置为 0，则代表无限换行，直到文字结束或者到达 UILabel 控件的最底端为止。

UILabel 中还有一个 `lineBreakMode` 属性，这个属性可以设置换行和截断模式，这个属性设置的值为 `NSLineBreakMode` 枚举，意义如下：

```
typedef NS_ENUM(NSInteger, NSLineBreakMode) {
    NSLineBreakByWordWrapping = 0,           // 以字符为标准换行
    NSLineBreakByCharWrapping,                // 以单词为标准换行
    NSLineBreakByTruncatingHead,              // 头部截断
    NSLineBreakByTruncatingTail,              // 尾部截断
    NSLineBreakByTruncatingMiddle             // 中间截断
} NS_ENUM_AVAILABLE(10_0, 6_0);
```

上面这个枚举值的设置效果如图 2-6~图 2-10 所示。



图 2-5 文字被截断的 UILabel 控件

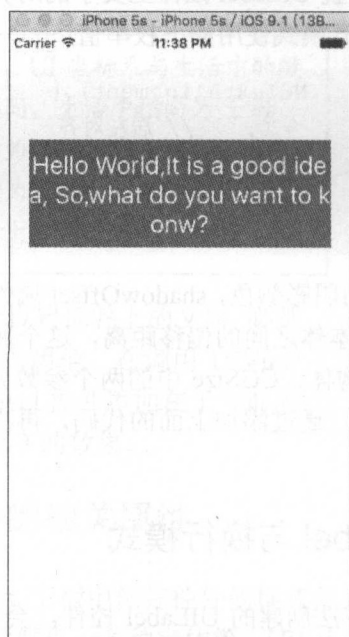


图 2-6 NSLineBreakBy WordWrapping

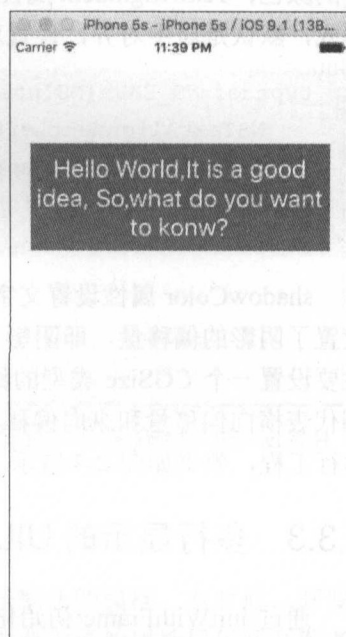


图 2-7 NSLineBreakBy CharWrapping

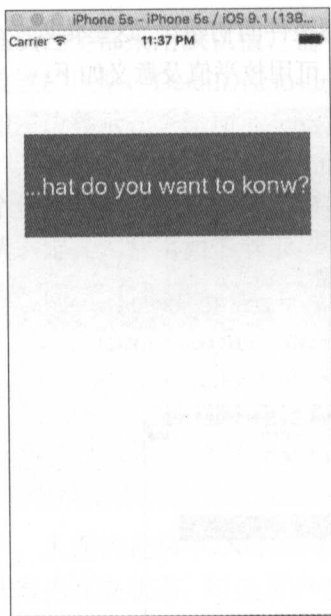


图 2-8 NSLineBreakBy
TruncatingHead

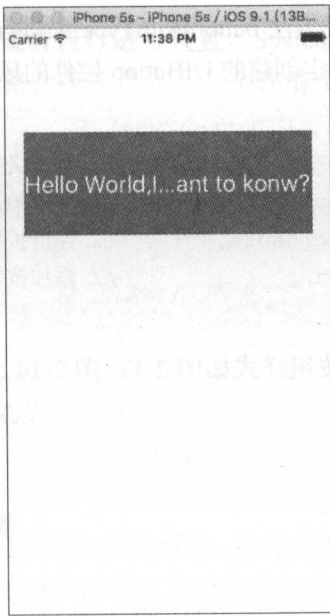


图 2-9 NSLineBreakBy
TruncatingTail

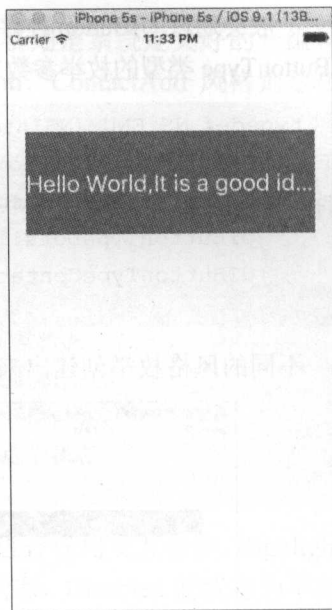


图 2-10 NSLineBreakBy
TruncatingMiddle

2.4 按钮控件——UIButton

对于一个应用 App 来说，展示和交互就是其生命。UILabel 控件可以说是 UIKit 框架中最简单基础的显示控件，与之对应的 UIButton 控件是 UIKit 框架中最简单基础的交互控件。UIButton 通常又被称为按钮控件，它的确也发挥着一个按钮的功能，可以监听到用户在屏幕视图上的多种手势操作。

2.4.1 创建一个按钮来改变屏幕颜色

通过 Xcode 创建一个名为 UIButtonTest 的工程，在 ViewController.m 的 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIButton * button = [UIButton buttonWithType:UIButtonTypeSystem];
    button.frame = CGRectMake(40, 100, 240, 30);
    button.backgroundColor = [UIColor redColor];
    [button setTitle:@"点我一下" forState:UIControlStateNormal];
    [button addTarget:self action:@selector(changeColor) forControlEvents:
    UIControlEventTouchUpInside];
    [self.view addSubview:button];
}
```


一般会通过 UIButton 控件的类方法 buttonWithType:来进行按钮控件的初始化, 这里传入一个 UIButtonType 类型的枚举参数来确定创建的 UIButton 控件的风格, 可用枚举值及意义如下:

```
typedef NS_ENUM(NSInteger, UIButtonType) {
    UIButtonTypeCustom = 0,           // 自定义类型
    UIButtonTypeSystem,               // 标准的系统类型
    UIButtonTypeDetailDisclosure,      // 详情按钮类型
    UIButtonTypeContactAdd,           // 添加按钮类型
};
```

不同的风格枚举创建出来的按钮样式如图 2-11~图 2-14 所示。

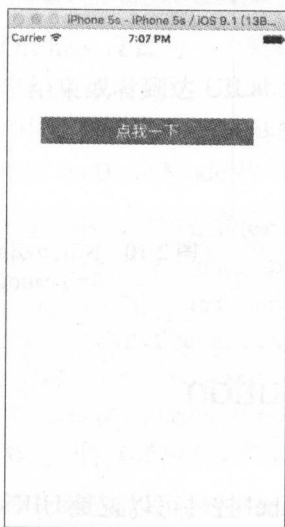


图 2-11 UIButtonTypeCustom

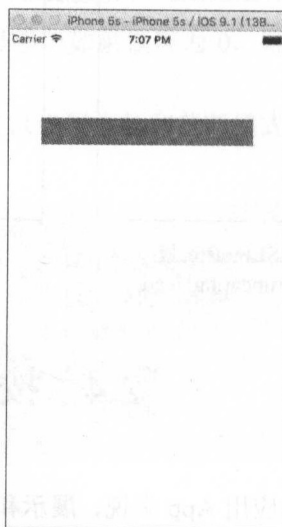


图 2-12 UIButtonTypeSystem

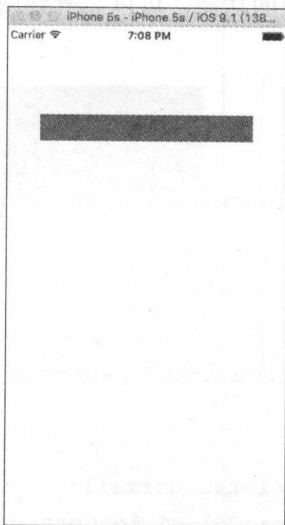


图 2-13 UIButtonTypeDetailDisclosure

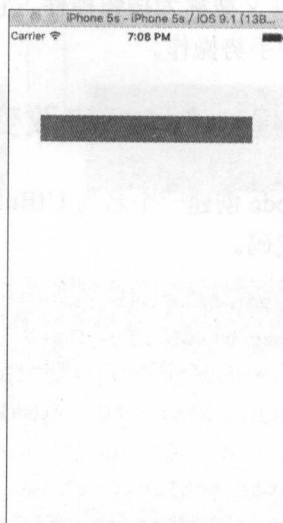


图 2-14 UIButtonTypeContactAdd

UIButton 控件也包括背景色、字体颜色、字体、单击状态等属性, Custom 风格则是将这

些属性全部采用默认值，需要开发者自行进行设置，System 风格则是系统定义好的一组属性设置的风格，DetailDisclosure 风格会在左边显示一个详情小 icon，ContactAdd 风格则是在按钮左边显示一个添加小 icon。

回到上面的代码，setTitle:forState:方法有两个参数，第一个设置了按钮的标题文字，第二个参数则是设置显示此标题文字时的按钮状态，UIControlState 中枚举了许多用于交互控件的状态定义，常用如下所示：

```
typedef NS_OPTIONS(NSUInteger, UIControlState) {
    UIControlStateNormal        = 0,           // 正常状态
    UIControlStateHighlighted    = 1 << 0,     // 高亮状态
    UIControlStateDisabled       = 1 << 1,     // 不可用状态
    UIControlStateSelected       = 1 << 2,     // 选中状态
};
```

上面的枚举中，Normal 状态是按钮的初始状态，此时并没有进行任何交互操作，Highlighted 状态为高亮状态，即当用户手指单击到按钮但并没有抬起时的状态，Disabled 的状态为不可用状态，此时用户的单击操作将无效，Selected 为选中状态，用于一些充当切换开关的按钮。

UIButton 控件的核心功能是进行用户交互，通过 addTarget:action:forControlEvents:方法进行触发方法的添加，这个方法需要 3 个参数，第 1 个参数为执行此触发方法的对象，一般会填写当前类对象 self，第 2 个参数为对应的方法，第 3 个参数为触发方法的条件，这里需要传入一个 UIControlEvents 的枚举对象，其中常用枚举值的意义如下所示：

```
typedef NS_OPTIONS(NSUInteger, UIControlEvents) {
    UIControlEventTouchDown,           // 用户手指按下时触发方法
    UIControlEventTouchDownRepeat,     // 用户多次重复按下时触发
    UIControlEventTouchDragInside,     // 用户在控件范围内进行拖滑移动时触发
    UIControlEventTouchDragOutside,    // 用户在控件范围内按下并且拖滑到控件范围外时触发
    UIControlEventTouchDragEnter,      // 用户手指拖动进控件范围后触发
    UIControlEventTouchDragExit        // 用户手指拖动结束时触发，
    UIControlEventTouchUpInside        // 用户在控件范围内按下并且在范围内抬起触发，即单击
    UIControlEventTouchUpOutside,      // 用户在控件范围内按下并在范围外抬起触发
    UIControlEventTouchCancel,         // 单击事件被取消时触发
    UIControlEventValueChanged,        // 控件的 value 值改变时触发
};
```

上面的这些枚举选项决定了触发交互的用户操作行为。

下面是实现 changeColor 触发方法的示例。

```
-(void)changeColor{
    self.view.backgroundColor = [UIColor colorWithRed:arc4random()%255/255.0 green:arc4random()%255/255.0 blue:arc4random()%255/255.0 alpha:1];
}
```

changeColor 方法中使用了通过 RGBA 方式来创建颜色对象，前 3 个参数分别设置了颜色红、绿、蓝 3 色值，第 4 个参数设置了颜色的透明度值，每个参数的取值都为 0~1 之间的浮点值。

运行工程，单击视图上的按钮，可以看到视图颜色随机切换的霓虹效果。

2.4.2 更加多彩的 UIButton 控件

在 2.4.1 节的例子中，只是创建了一个未加任何修饰的按钮控件，可以通过一些 UIButton 的一些属性，为其添加背景或者内容图片来创建更多彩的按钮。

首先，先向工程项目中添加一张图片，在工程的导航窗口部分中单击 Assets.xcassets 包文件，选择一张图片将其拖入素材区，如图 2-15 所示。

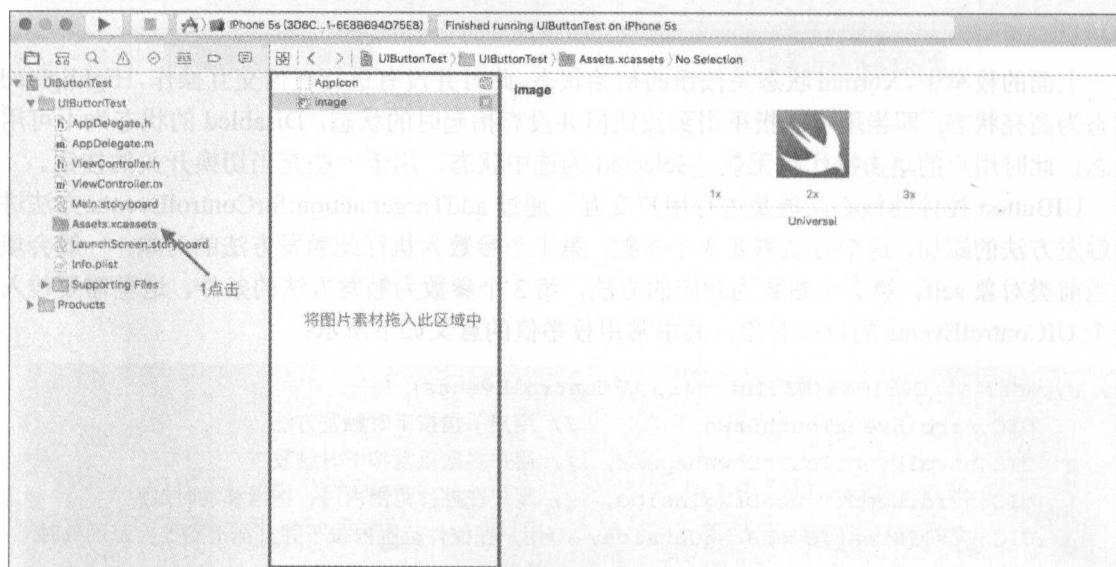


图 2-15 向工程中添加图片素材

通过修改为下面的代码来对按钮控件进行相关设置：

```
UIButton * button = [UIButton buttonWithType:UIButtonTypeCustom];
[button setBackgroundImage:[UIImage imageNamed:@"image"] forState:UIControlStateNormal];
```

运行工程，会看到如图 2-16 所示，按钮被添加上了图片背景。

可以发现，背景图片的效果是当图片作为背景时，按钮标题将显示在图片层之上，UIButton 中还有一个方法用于设置图片为内容图片，这种情况下，图片将和标题并列显示，代码如下所示。

```
[button setImage:[UIImage imageNamed:@"image"] forState:UIControlStateNormal];
```

效果如图 2-17 所示。

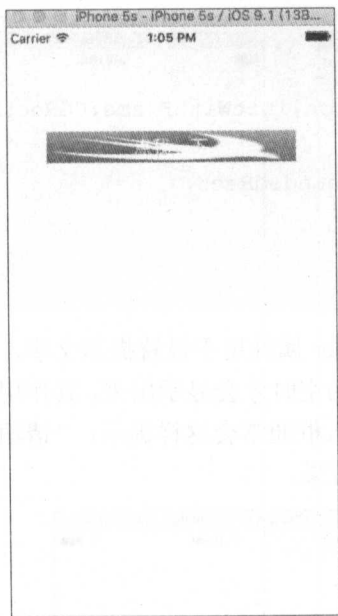


图 2-16 按钮添加背景图片

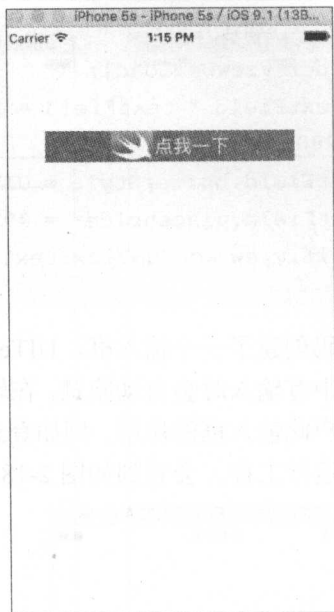


图 2-17 按钮添加内容图片

在图 2-17 中可以看到，图片和文字是左右并排排列并且共同居中显示的。在很多时候，开发者会需要上下排列或者其他位置排列的布局方式，UIButton 类中也提供了接口供开发者进行内容、图片和文字的位置设置，示例代码如下：

```
[button setContentEdgeInsets:UIEdgeInsetsMake(0, 0, 0, 0)];
[button setImageEdgeInsets:UIEdgeInsetsMake(0, 0, 0, 0)];
[button setTitleEdgeInsets:UIEdgeInsetsMake(0, 0, 0, 0)];
```

setContentEdgeInsets:方法用于设置整体内容的区域偏移量，UIEdgeInsetsMake()方法中的 4 个参数分别代表上、左、下、右 4 个方向的偏移量。setImageEdgeInsets:方法只设置内容图片的位置偏移量，setTitleEdgeInsets:方法只设置内容标题的位置偏移量。

2.5 文本输入框控件——UITextField

相比 UILabel 和 UIButton 控件，UITextField 控件要复杂的多，UITextField 是 iOS 系统中进行文本输入操作的一种 UI 控件，用户通过键盘将输入操作传递给 UITextField 控件，UITextField 控件采用代理的设计模式，再将用户的一些操作行为以回调方式传递给开发者，最后由开发者来进行具体的逻辑处理。

2.5.1 在屏幕上创建一个输入框

打开 Xcode，创建一个名为 UITextFieldTest 的工程，在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码。


```

- (void)viewDidLoad {
    [super viewDidLoad];
    UITextField * textField = [[UITextField alloc] initWithFrame:CGRectMake(
(20, 100, 280, 30)];
    textField.borderStyle = UITextBorderStyleRoundedRect;
    textField.placeholder = @"请输入文字";
    [self.view addSubview:textField];
}

```

上面代码创建了一个输入框，UITextField 的 placeholder 属性用于设置提示文字，这些文字在输入框中有输入时会自动隐藏，在输入框输入的内容为空时才会显示出来，其作用主要是用来提示用户此输入框的作用，例如登录界面的用户名输入框通常会这样提示：“请输入您的用户名”。运行工程，会看到如图 2-18 和图 2-19 所示的效果。

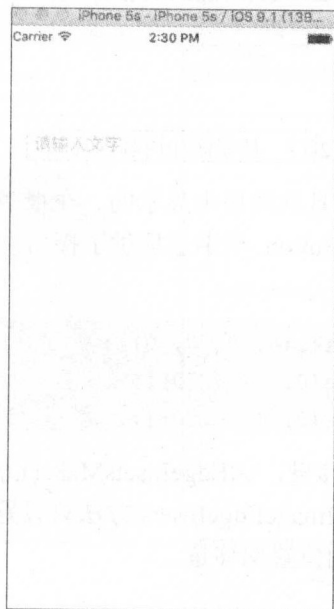


图 2-18 未输入文字的输入框

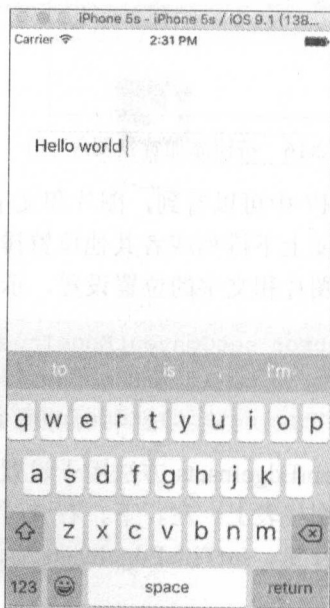


图 2-19 正在输入的输入框

UITextField 的 borderStyle 属性用于设置输入框的界面风格，UITextBorderStyle 枚举值的意义如下：

```

typedef NS_ENUM(NSInteger, UITextBorderStyle) {
    UITextBorderStyleNone,           //无风格
    UITextBorderStyleLine,          //线性风格
    UITextBorderStyleBezel,         //bezel 风格
    UITextBorderStyleRoundedRect    //边框风格
};

```

边框风格的界面效果如图 2-18 所示，其他风格的输入框界面效果如图 2-20~图 2-22 所示。



图 2-20 UITextBorderStyleNone



图 2-21 UITextBorderStyleLine



图 2-22 UITextBorderStyleBezel

2.5.2 UITextField 的常用属性介绍

UITextField 中也有相关属性用于输入框中文字属性的设置，代码示例如下：

```
//设置输入框中文字颜色
textField.textColor = [UIColor redColor];
//设置输入框字体
textField.font = [UIFont systemFontOfSize:14];
//设置文字的对齐模式
textField.textAlignment = NSTextAlignmentCenter;
```

开发者除了对输入框中文字属性进行设置之外，UITextField 还支持自定义左视图和右视图。UITextField 的左视图应用十分广泛，例如很多密码输入框的左边都会有一个小钥匙的图片，用来提示用户输入框的作用。设置左视图的代码示例如下：

```
UIImageView * imageView = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"image"]];
textField.leftView = imageView;
textField.leftViewMode = UITextFieldViewModeAlways;
```

UITextField 的 leftView 属性需要传入一个 UIView 或者其子类的对象，示例代码中使用了 UIImageView，leftViewMode 属性设置了显示左视图的显示模式，枚举意义如下：

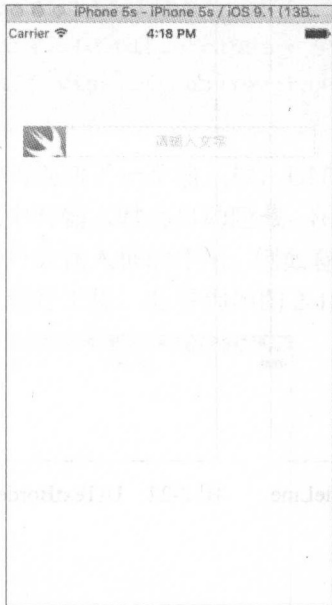
```
typedef NS_ENUM(NSInteger, UITextFieldViewMode) {
    UITextFieldViewModeNever,           //从不显示
    UITextFieldViewModeWhileEditing,    //编辑时显示
    UITextFieldViewModeUnlessEditing,   //非编辑时显示
};
```

```
UITextFieldViewModeAlways
```

```
//总是显示
```

```
};
```

此时运行工程，可以看到如图 2-23 的界面效果。



提示

UIImageView 是专门用来展示图片的视图类。其不仅可以通过设置 image 属性进行图片的渲染展示,还可以通过一些设置实现帧动画的播放。

图 2-23 显示左视图的 UITextField

2.5.3 UITextField 的代理方法

在本章的开头部分，介绍了代理这种设计模式，UITextField 的一些回调就是通过代理来实现的。例如很多网站的会员账号是采用手机号来注册的，这时对于用户名输入框来说，其只能允许用户输入不超过 11 位的数字，如果用户输入非数字字符或者输入超限，应用会进行处理，使用户的这次输入无效。其实这个过程就是一个代理回调的过程，首先用户输入一个字符，字符被传进 UITextField 中，UITextField 无法判断这个字符是否有效，它将字符通过代理再传递给开发者，开发者来做逻辑处理。UITextFieldDelegate 中支持以下这些代理方法：

```
- (BOOL)textFieldShouldBeginEditing:(UITextField *)textField; //当输入框将要开始编辑时系统自动回调的方法
- (void)textFieldDidBeginEditing:(UITextField *)textField; //当输入框已经开始编辑时系统自动回调的方法
- (BOOL)textFieldShouldEndEditing:(UITextField *)textField; //输入框将要结束编辑模式时系统自动回调的方法
- (void)textFieldDidEndEditing:(UITextField *)textField; //输入框已经结束编辑模式时系统自动回调的方法
- (BOOL)textField:(UITextField *)textField shouldChangeCharactersInRange:(NSRange)range replacementString:(NSString *)string; // 输入框中内容将要改变时系统自动回调的方法
- (BOOL)textFieldShouldClear:(UITextField *)textField; //输入框中内容将
```

要被清除时系统自动回调的方法

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField;    //用户单击键盘 re
turn 键后系统自动回调的方法
```

textFieldShouldBeginEditing:方法是当用户在屏幕上单击输入框，键盘将要弹出来时会调用。这个函数有一个 BOOL 类型的返回值，如果开发者在实现这个函数时返回 NO，则键盘不会弹出来，UITextFiled 控件也不能进入编辑状态。**textFiedShouldEndEditing:**方法与 **textFieldShouldBeginEditing:**方法类似，只是它对应的是结束编辑状态。**textFieldDidBeginEditing:**方法和 **textFieldDidEndEditing:**方法分别是在 UITextField 已经开始和已经结束编辑状态时触发的方法。**shouldChangeCharactersInRange:replacementString:**方法在输入框中内容将要改变时会调用，其内会传进两个参数给开发者使用，**range**是将要改变的字符范围，**string**是将要替换成的字符串，同时这个函数还需要返回一个 BOOL 类型的返回值，如果返回 NO，则这次字符改变行为则不成功，判断用户的输入是否合法的操作，一般会放在这个代理方法中进行。**textFieldShouldClear:**方法在单击清除按钮后会被调用，这里的返回值如果返回 NO，则这次清除操作无效。**textFieldShouldReturn:**方法在用户单击键盘上的 **return** 按钮后进行调用。



提示

所谓 UITextField 的编辑状态，就是光标出现在输入框中并且闪烁，键盘弹出等待用户输入的状态。

2.5.4 实现一个监听输入信息的用户名输入框

上面介绍了 UITextFieldDelegate 中的相关方法，有了这些知识，已经可以实现一个实时监听的输入框了，要使用代理方法需要 3 个步骤：

- (1) 遵守相应协议
- (2) 设置代理
- (3) 实现代理方法

首先，在类的声明部分添加要遵守的代理，如下所示。

```
@interface ViewController ()<UITextFieldDelegate>
@end
```

在 **viewDidLoad** 方法中添加如下一行代码进行代理的设置。

```
textField.delegate = self;
```

在 **ViewController.m** 文件中实现如下的代理方法。

```
- (BOOL)textField:(UITextField *)textField shouldChangeCharactersInRange:
(NSRange)range replacementString:(NSString *)string{
    if (string.length>0) {
        if ([string characterAtIndex:0]<'0' || [string characterAtIndex:0]>'9')
        {
```



```

        NSLog(@"请输入数字");
        return NO;
    }

    if (textField.text.length+string.length>11) {
        NSLog(@"超过 11 位啦");
        return NO;
    }

    return YES;
}

```

在实现的 `textField:shouldChangeCharactersInRange:replacementString:` 方法中, 先进行了是否是数字的逻辑判断。如果不是数字, 则会打印提示信息, 并且使本次输入无效, 之后判断数字是否超过 11 位, 这个判断条件中取的字符长度是输入框上原有文字的长度加上本次输入的字符的长度, 如果超过 11 位, 进行打印信息并使本次输入无效。这样, 一个只能输入数字且不可超过 11 位的输入框就编写完成了, 如图 2-24 所示。

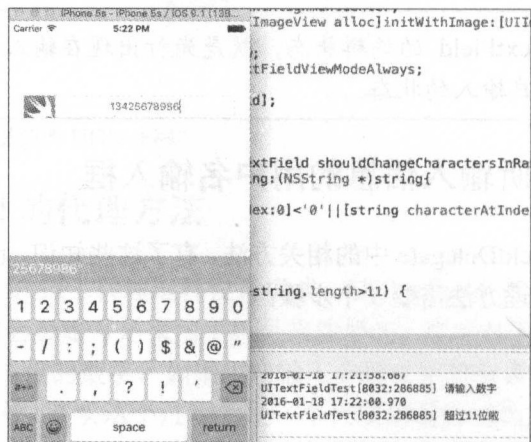


图 2-24 监听用户输入操作的 UITextField

2.6 开关控件——UISwitch

UISwitch 是 UIKit 框架中的一个十分小巧简洁的控件, 其用于一些简单的切换功能逻辑中, 在很多 Apple 自行开发的应用中, 这个控件的使用率也是非常高的。

2.6.1 创建一个开关控件

使用 Xcode 创建一个名为 UISwitchTest 的工程, 在 ViewController.m 的 viewDidLoad 方法中添加如下代码。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    UISwitch * swi = [[UISwitch alloc] initWithFrame:CGRectMake(100, 100, 100, 40)];
    swi.onTintColor = [UIColor greenColor];
    swi.tintColor = [UIColor redColor];
    swi.thumbTintColor = [UIColor orangeColor];
    [self.view addSubview:swi];
}

```

UISwitch 的功能十分简单，因此其可设置的属性也十分有限，onTintColor 属性用于设置控件开启状态的填充色，tintColor 属性设置控件关闭状态的边界色，thumbTintColor 属性设置开关按钮的颜色，运行工程，效果如图 2-25 和图 2-26 所示。

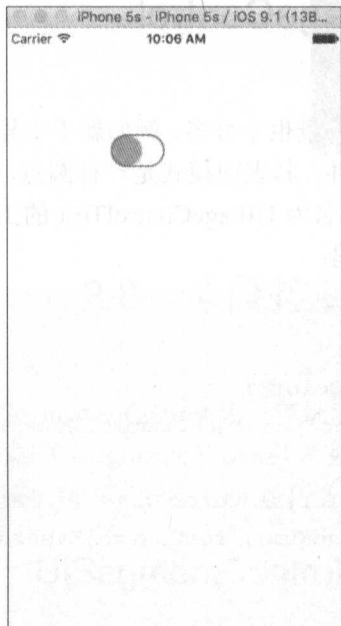


图 2-25 关闭状态的 UISwitch 空间

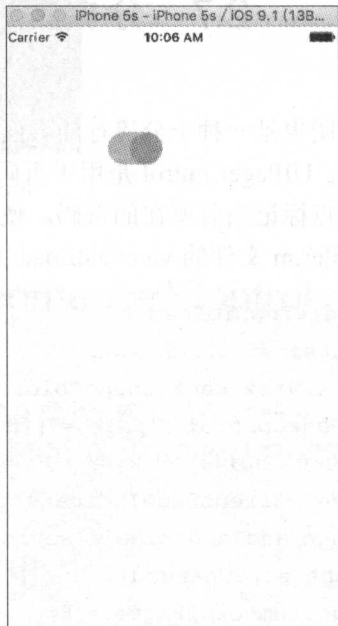


图 2-26 开启状态的 UISwitch 控件

2.6.2 为 UISwitch 控件添加触发方法

UISwitch 也属于用户交互控件，可以为其添加交互方法来处理某些开与关的逻辑。UISwitch 继承于 UIControl，继承于 UIControl 的类都可以通过 addTarget:action:forControlEvents: 方法来进行触发方法的添加，代码如下：

```

[swi addTarget:self action:@selector(changeColor:) forControlEvents:UIControlEventValueChanged];

```

这里实现的触发方法使用带一个参数值的函数，系统传入的参数即为 UISwitch 对象本身，方法实现代码如下：

```

-(void)changeColor:(UISwitch *)swi{
    if (swi.isOn) {
        self.view.backgroundColor = [UIColor redColor];
    }else{
        self.view.backgroundColor = [UIColor whiteColor];
    }
}

```

UISwitch 的 `isOn` 属性是一个 `BOOL` 值，通过这个值可以判断 UISwitch 控件的开关状态，然后分别进行相应的操作即可，这里在切换 UISwitch 控件的开关状态时进行了当前视图背景颜色的转换。

2.7 分页控制器——UIPageControl

分页视图也是一种十分流行的界面设计模式，使用的场景也十分多，例如新手引导页和广告轮播页等。UIPageControl 是用于页码管理的一个 UI 控件，其表现模式是一行圆点，其中高亮的一个圆点标记当前所在的页码。使用 Xcode 创建一个名为 `UIPageControlTest` 的工程，在 `ViewController.m` 文件的 `viewDidLoad` 方法中添加如下代码：

```

- (void)viewDidLoad {
    [super viewDidLoad];
    self.view.backgroundColor = [UIColor blackColor];
    UIPageControl * page = [[UIPageControl alloc] initWithFrame:CGRectMake(
20, 100, 280, 30)];
    page.currentPageIndicatorTintColor = [UIColor redColor];
    [page addTarget:self action:@selector(changeNum:) forControlEvents:UI
ControlEventValueChanged];
    page.numberOfPages = 8;
    [self.view addSubview:page];
}

```

将当前视图的背景颜色设置为黑色便于进行效果的演示，UIPageControl 的 `currentPageIndicatorTintColor` 属性设置高亮页码点的颜色，`numberOfPages` 属性设置页码数量，运行工程，效果如图 2-27 所示。

在单击 UIPageControl 控件的左半边时，页码点会向左移动，单击 UIPageControl 的右半边时，页码点会向右移动。在单击 UIPageControl 控件的同时也会触发交互方法，交互方法的实现如下所示，这里打印了当前的页码数（从 0 开始）。

```

-(void)changeNum:(UIPageControl *)page{
    NSLog(@"%lu",page.currentPage);
}

```

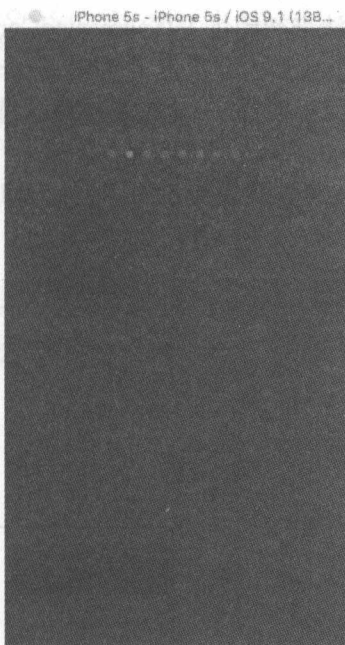


图 2-27 UIPageControl

2.8 分段控制器——UISegmentedControl

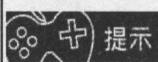
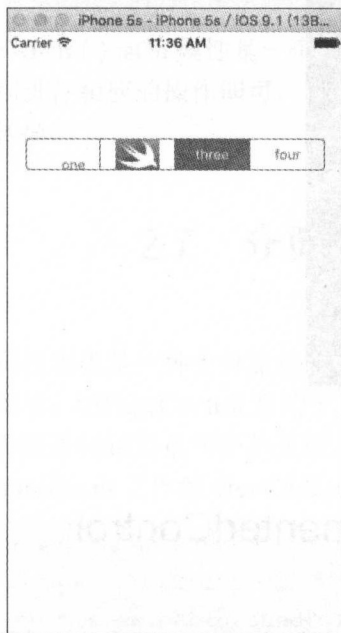
UISegmentedControl 用于管理和实现一组内容的切换逻辑,例如几个并列关系的界面之间相互切换。UISegmentedControl 常见于导航栏的标题视图中,因其小巧的外表和简洁的接口风格,在原生和第三方应用中都十分常见。

2.8.1 UISegmentedControl 基本属性的应用

首先使用 Xcode 创建一个名为 UISegmentedControlTest 的工程,在 ViewController.m 的 viewDidLoad 方法中添加如下代码:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UISegmentedControl * seg = [[UISegmentedControl alloc] initWithItems:@
    [@"one",@"",@"three",@"four"]];
    seg.frame = CGRectMake(20, 100, 280, 30);
    [seg setImage:[UIImage imageNamed:@"image"] imageWithRenderingMode:U
    IImageRenderingModeAlwaysOriginal forSegmentAtIndex:1];
    [seg setContentOffset:CGSizeMake(10, 10) forSegmentAtIndex:0];
    seg.momentary=NO;
    [self.view addSubview:seg];
}
```


代码中使用 `initWithItems:` 方法来初始化 `UISegmentedControl` 对象，这个方法中需要传入一个标题数组，数组中字符串的个数和内容决定了 `UISegmentedControl` 中按钮的个数和标题。`setImage:forSegmentAtIndex:` 方法用于设置某个按钮的图案，其中按钮的 `Index` 从 0 开始计。`setContentOffset:forSegmentAtIndex:` 方法设置其中某个按钮内容的位置偏移。`UISegmentedControl` 的 `momentary` 属性默认为 `NO`，控件为切换按钮模式，如果设置为 `YES`，则控件为触发按钮模式。运行工程，效果如图 2-28 所示。



提示

`UISegmentedControl` 在切换按钮模式时，当用户点击一个按钮后，此按钮会一直保持选中状态直到用户切换为另一个为止，而在触发按钮模式中，用户手指离开屏幕后，按钮将不保持选中状态。

图 2-28 `UISegmentedControl`

2.8.2 对 `UISegmentedControl` 中的按钮进行增、删、改操作

`UISegmentedControl` 对象中的按钮除了在初始化时可以进行创建外，在程序运行过程中，也可以进行动态的添加、删除和修改操作，`UISegmentedControl` 中有如下方法可供开发者使用。

```
[seg insertSegmentWithTitle:@"new" atIndex:2 animated:YES];
[seg removeSegmentAtIndex:1 animated:YES];
[seg setTitle:@"replace" forSegmentAtIndex:1];
[seg removeAllSegments];
```

`insertSegmentWithTitle:atIndex:animated:` 方法用于在 `UISegmentedControl` 当前按钮数量的前提下插入一个新的标题按钮，第 1 个参数是设置按钮的标题文字，第 2 个参数是设置插入的位置，第 3 个参数是设置是否带动画效果，与这个方法对应的还有一个：`insertSegmentWithImage:atIndex:animated:` 方法用于插入一个图片按钮。`removeSegmentAtIndex:animated:` 方法可以在已有的按钮中移除一个。`setTitle:forSegmentAtIndex:` 方法可以重新设置一个按钮的标题，与之对应的 `setImage:forSegmentIndex:` 方法可以重新设置一个按钮的图片。`removeAllSegments` 方法将移除所有的按钮。

2.8.3 UISegmentedControl 中按钮宽度的自适应

UISegmentedControl 控件其中的按钮宽度默认是平均分的,如果某个按钮的标题长度超出了宽度的范围,将会被自动截断,如图 2-29 所示。

开发者可以手动对 UISegmentedControl 中的每个按钮的宽度进行设置,以便设置按钮宽度与其文字相适应,示例代码如下,效果如图 2-30 所示。

```
[seg setWidth:130 forSegmentAtIndex:3];
```

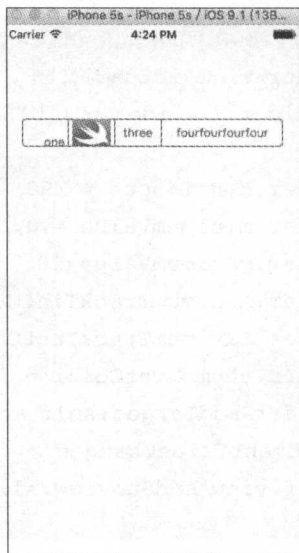
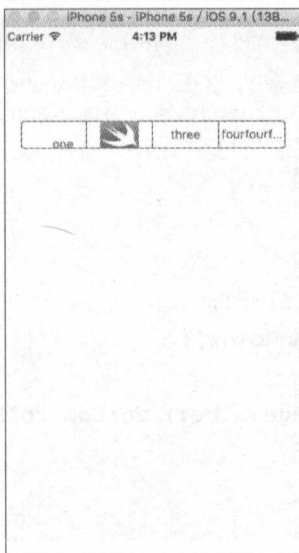


图 2-29 UISegmentedControl 按钮标题文字被截断

图 2-30 自定义 UISegmentedControl 按钮宽度

上面的方法可以做到 UISegmentedControl 中按钮宽度的设置,但是有一个致命的缺点,开发者或者并不知道这个按钮上面标题文字所占的宽度,如果使用强制计算的方法,又会徒增许多的代码量。幸运的是,UISegmentedControl 中还提供了一个属性,可以让 UISegmentedControl 自己计算其中按钮需要的宽度,让其进行宽度的自适应,代码如下所示:

```
seg.apportionsSegmentWidthsByContent=YES;
```

将 `apportionsSegmentWidthsByContent` 属性设置为 YES,则 UISegmentedControl 控件中的按钮宽度将进行自适应。



提示

UISegmentedControl 添加触发方法也是通过 `addTarget:action:forControlEvents:` 方法来设置的。

2.9 滑块控件——UISlider

在前几节所介绍的控件中,它们都有一个共同的特点,其状态的变化或者其值的变化都是

离散的几种，例如 UIButton 的正常、高亮、选中，UISwitch 的开和关，UISegmentedControl 的按钮值的切换等。UISlider 与上述控件最大的区别在于它值的变化可以是连续的，由于这种特性，UISlider 可以用于处理一些连续变化量的交互逻辑。

2.9.1 UISlider 的创建与常规设置

打开 Xcode，创建一个名为 UISliderTest 的工程，在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UISlider * slider = [[UISlider alloc] initWithFrame:CGRectMake(20, 100,
280, 30)];
    slider.continuous = YES;
    slider.minimumValue = 0;
    slider.maximumValue=10;
    slider.minimumTrackTintColor = [UIColor redColor];
    slider.maximumTrackTintColor = [UIColor greenColor];
    slider.thumbTintColor = [UIColor blueColor];
    [slider addTarget:self action:@selector(changeValue:) forControlEvents
s:UIControlEventsValueChanged];
    [self.view addSubview:slider];
}
```

continuous 属性用于设置 UISlider 控件的触发方法是否连续触发，设置为 YES，则用户在滑动滑块时，触发方法会多次执行，如果设置为 NO，则只有当用户滑动结束时，方法才会触发。minimumValue 设置 UISlider 控件的最小值，即滑块在最左端时控件的值。maximumValue 设置 UISlider 控件的最大值，即滑块在最右端时控件的值。minimumTrackTintColor 属性设置滑块以左中轴的颜色。maximumTrackColor 设置滑块以右中轴的颜色。thumbTintColor 设置滑块本身的颜色。运行工程，效果如图 2-31 所示。

使用 addTarget:action:forControlEvents: 进行触发方法的添加，在 UISlider 类型的参数中可以获取到控件的当前值并进行逻辑处理，示例如下所示。

```
-(void)changeValue:(UISlider *)slider{
    NSLog(@"%f", slider.value);
}
```

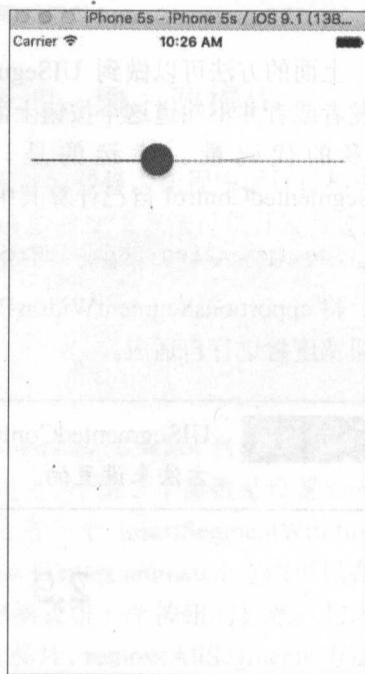


图 2-31 UISlider 控件

2.9.2 对 UISlider 添加图片修饰

UISlider 类提供了一些方法来对 UISlider 控件进行一些图片的修饰, 使用如下代码。

```
slider.minimumValueImage = [UIImage imageNamed:@"image"];
slider.maximumValueImage = [UIImage imageNamed:@"image"];
[slider setThumbImage:[UIImage imageNamed:@"image"] forState:UIControlStateNormal];
```

minimumValueImage 属性设置左视图图片, maximumValueImage 属性设置右视图图片, setThumbImage:forState:方法设置控件在某个状态下的滑块图片, 这时运行工程, 效果如图 2-32 所示。

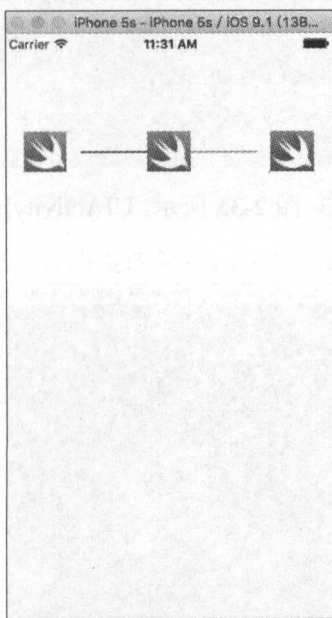


图 2-32 添加了图片修饰的 UISlider 控件

2.10 活动指示器控件——UIActivityIndicatorView

UIActivityIndicatorView 通常又被称为风火轮控件。在某些加载复杂视图, 下载数据的场景中经常可以看到它的身影。其主要作用是在加载等待的时间中给用户一些界面活动的提示, 不至于使用户感觉到界面卡死的假象。

使用 Xcode 开发工具创建一个名为 UIActivityIndicatorViewTest 的工程, 在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码:

```
- (void)viewDidLoad {
    [super viewDidLoad];
```



```

self.view.backgroundColor = [UIColor redColor];
UIActivityIndicatorView * indicator = [[UIActivityIndicatorView alloc]
initWithActivityIndicatorStyle:UIActivityIndicatorViewStyleGray];
indicator.center = CGPointMake(self.view.frame.size.width/2, self.view.frame.size.height/2);
indicator.color = [UIColor blackColor];
[self.view addSubview:indicator];
[indicator startAnimating];
}

```

`initWithActivityIndicatorStyle:`方法通过一个风格枚举来对控件进行初始化，`UIActivityIndicatorViewStyle` 中枚举的值意义如下：

```

typedef NS_ENUM(NSInteger, UIActivityIndicatorViewStyle) {
    UIActivityIndicatorViewStyleWhiteLarge,    //大号白色风格
    UIActivityIndicatorViewStyleWhite,        //白色风格
    UIActivityIndicatorViewStyleGray,         //灰色风格
};

```

其中各个风格的效果如图 2-33~图 2-35 所示。`UIActivityIndicatorView` 的 `color` 属性可以设置活动指示器的颜色。

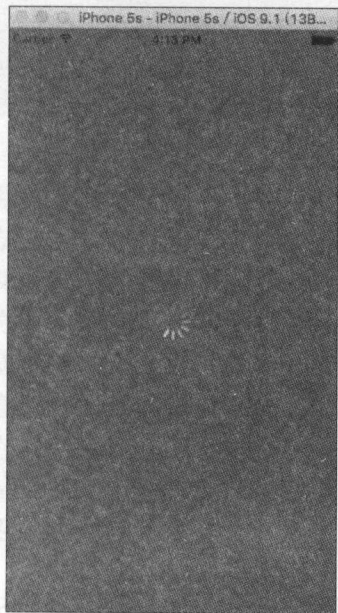


图 2-33 UIActivityIndicatorViewStyleWhiteLarge

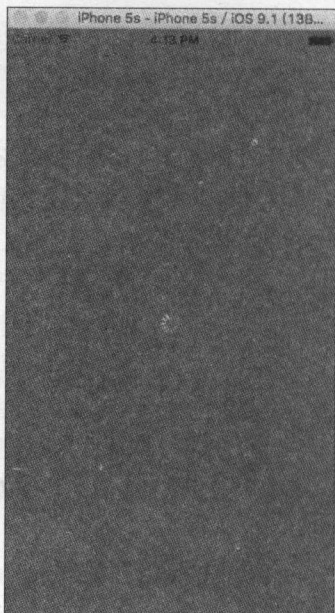


图 2-34 UIActivityIndicatorViewStyleWhite

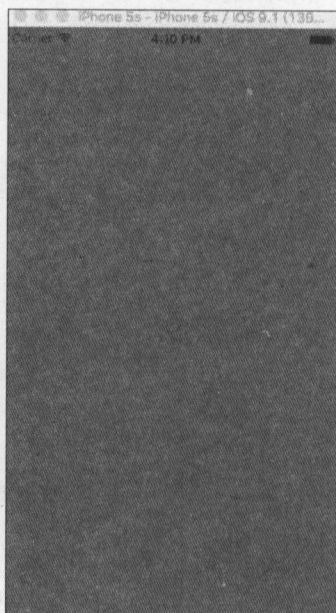


图 2-35 UIActivityIndicatorViewStyleGray

将活动指示器控件添加到视图上之后，需要调用 `startAnimating` 方法来使指示器开始转动，与之对应，调用 `stopAnimating` 方法来使指示器停止转动。

2.11 进度条控件——UIProgressView

UIKit 框架中的 `UIProgressView` 控件可以创建一个进度条, 这个控件在播放器类软件中较为常见, 使用 Xcode 创建一个名为 `UIProgressViewTest` 的工程, 在 `ViewController.m` 文件的 `viewDidLoad` 方法中添加如下代码:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIProgressView * progressView = [[UIProgressView alloc] initWithFrame:CGRectMake(20, 100, 280, 30)];
    progressView.progressTintColor = [UIColor redColor];
    progressView.trackTintColor = [UIColor blueColor];
    [self.view addSubview:progressView];
    progressView.progress = 0.5;
}
```

`progressTintColor` 属性设置已走过进度的颜色, `trackTintColor` 属性设置未走过的进度的颜色, `progress` 属性设置进度条当前的进度, 取值为 0~1 之间的浮点数。运行上面代码后, 效果如图 2-36 所示。

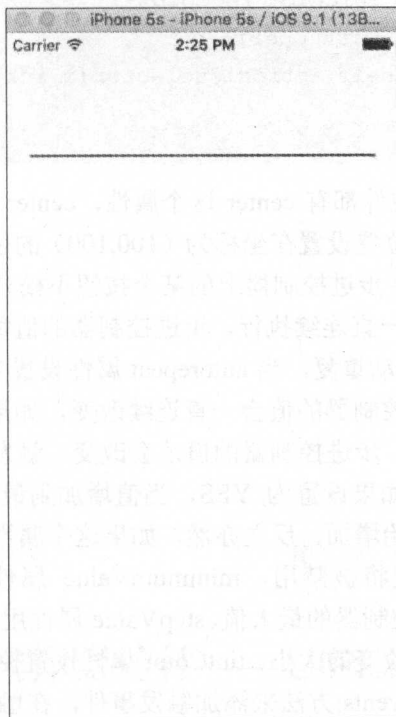


图 2-36 UIProgressView 控件

2.12 步进控制器——UIStepper

步进控制器，从名字就可以了解，其功能是进行离散式的数据调节。

2.12.1 步进控制器的基本属性使用

使用 Xcode 创建一个名为 UIStepperTest 的工程，在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIStepper * stepper = [[UIStepper alloc] init];
    stepper.center = CGPointMake(100, 100);
    stepper.continuous = YES;
    stepper.autorepeat = YES;
    stepper.wraps = YES;
    stepper.minimumValue = 1;
    stepper.maximumValue = 10;
    stepper.stepValue = 1;
    stepper.tintColor = [UIColor redColor];
    [self.view addSubview:stepper];
    [stepper addTarget:self action:@selector(click:) forControlEvents:UIControlEventValueChanged];
}
```

UIKit 框架中所有的视图控件都有 center 这个属性，center 属性用于设置控件的中心位置坐标，上面代码将步进控制器位置设置在坐标为 (100,100) 的位置。continuous 属性设置触发方法是否连续执行，当用户按住步进控制器上的某个按钮不松开时，如果 continuous 属性设置为 YES，则添加的触发方法会一直连续执行，步进控制器的值每变化一次方法都会执行一次。autorepeat 属性从字面理解为自动重复，当 autorepeat 属性设置为 YES 时，用户如果按住步进控制器中的按钮不放，则步进控制器的值会一直连续改变，如果 autorepeat 设置为 NO，则直到用户手指抬起完成单击动作，步进控制器的值才会改变，触发方法才会执行。wraps 属性设置步进控制器的值是否循环，如果设置为 YES，当值增加到最大时，如果用户继续单击增加按钮，则值会从最小值重新开始增加，反之亦然，如果这个属性设置为 NO，则当步进控制器到达极值的时候，相应的按钮将被禁用。minimumValue 属性设置步进控制器的最小值，maximumValue 属性设置步进控制器的最大值。stepValue 属性用于设置步进控制器的步长及每次按下按钮后步进控制器的值改变的大小。tintColor 属性设置控件的颜色。步进控制器也是通过 addTarget:action:forControlEvents:方法来添加触发事件，在触发方法中会传入 UIStepper 对象本身，开发者通过获取其值来做相应的逻辑处理，上面代码中的 click:方法实现如下，这里打印了 UIStepper 控件的值。


```

- (void)click: (UIStepper *)step{
    NSLog(@"%f", step.value);
}

```

运行上面的程序代码，会看到如图 2-36 所示的效果。

2.12.2 自定义 UIStepper 按钮图片

在图 2-37 中可以看到，系统的 UIStepper 控件默认是显示一个减号和一个加号，单击加号值增加，单击减号值减小，开发者也可以通过以下方法自定义两个按钮的图片。

```

[stepper setDecrementImage:[UIImage imageNamed:@"image"] imageWithRe
nderingMode:UIImageRenderingModeAlwaysOriginal] forState:UIControlStateNormal];

```

```

[stepper setIncrementImage:[UIImage imageNamed:@"image"] imageWithRe
nderingMode:UIImageRenderingModeAlwaysOriginal] forState:UIControlStateNormal];

```

setDecrementImage:imageWithRenderingMode:forState: 方法设置减按钮的图片，setIncrementImage:imageWithRenderingMode:forState:方法设置加按钮的图片。效果如图 2-38 所示。

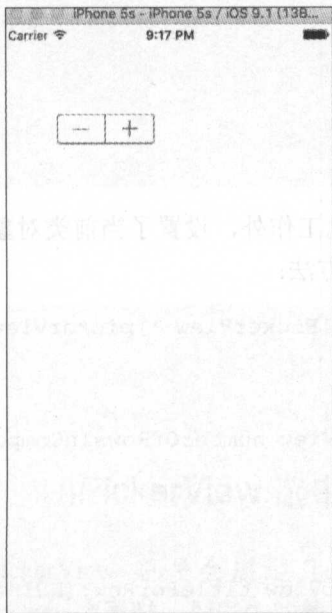


图 2-37 UIStepper 控件

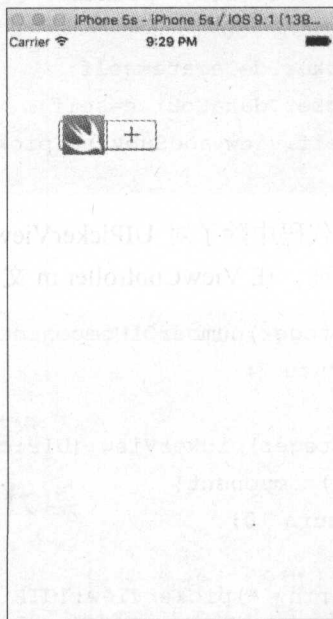


图 2-38 自定义图片的 UIStepper

2.13 选择器控件——UIPickerView

UIPickerView 是一个简易的列表控件，用于提供给用户有限个数的选项供用户选择，

UIPickerView 的 UI 设计十分漂亮，是 iOS 系统特有的 UI 模式。在实际应用中，UIPickerView 的应用也十分广泛，例如省市县选择列表、时间选择、日期选择等都可以通过 UIPickerView 来设计。

2.13.1 创建一个 UIPickerView 控件

UIPickerView 与之前章节中的 UI 控件有着很大不同，UIPickerView 更加复杂一些，它是通过代理和数据源的方法来对其进行设置和数据源的填充，这种控件的设计模式也是代理模式的应用之一。使用 Xcode 创建一个名为 UIPickerViewTest 的工程，在 ViewController.m 文件的声明部分添加对相应协议的遵守，这里需要遵守两个协议，分别是 UIPickerViewDelegate 和 UIPickerViewDataSource。

```
@interface ViewController () <UIPickerViewDataSource, UIPickerViewDelegate>
@end
```

在 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIPickerView * picker = [[UIPickerView alloc] initWithFrame:CGRectMake(
(20, 100, 280, 150)];
    picker.delegate=self;
    picker.dataSource=self;
    [self.view addSubview:picker];
}
```

上面的代码中除了对 UIPickerView 进行创建和初始化工作外，设置了当前类对象为它的数据源和代理。在 ViewController.m 文件中实现如下代理方法：

```
-(NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView{
    return 2;
}
-(NSInteger)pickerView:(UIPickerView *)pickerView numberOfRowsInComponent:
(NSInteger)component{
    return 10;
}
-(NSString *)pickerView:(UIPickerView *)pickerView titleForRow:(NSInteger)
row forComponent:(NSInteger)component{
    return [NSString stringWithFormat:@"%lu 分区%lu 数据", component, row];
}
-(CGFloat)pickerView:(UIPickerView *)pickerView heightForComponent:(NS
Integer)component{
    return 44;
}
```

```

-(CGFloat)pickerView:(UIPickerView *)pickerView widthForComponent:(NSInteger) component{
    return 140;
}

```

`numberOfComponentsInPickerView:` 方法返回一个 `NSInteger` 类型的整数，用于设置 `UIPickerView` 视图的分区数，也可以理解为选择列表的列数。

`pickerView:numberOfRowsInComponent:` 方法的返回值设置 `UIPickerView` 每个分区的行数，参数 `component` 用于判断具体的分区。

`pickerView:titleForRow:forComponent:` 方法的返回值设置列表中每一行的数据，这个方法中的两个参数 `row` 和 `component` 分别用于区分行与列。

`pickerView:rowHeightForComponent:` 方法的返回值设置具体行的行高。

`pickerView:widthForComponent:` 方法的返回值设置分区的宽度，即列的宽度。

运行工程，效果如图 2-39 所示。

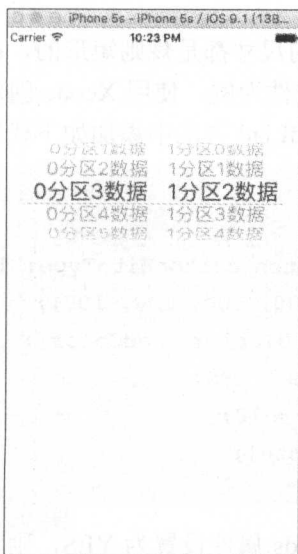


图 2-39 UIPickerView 控件

2.13.2 UIPickerView 选中数据时的回调代理

`UIPickerView` 总是会展现几列数据，帮助用户进行快速选择，当用户上下滑动 `UIPickerView` 列表时，列表中的数据会进行上下滑动移动，当移动动作停止时，这时悬停在 `UIPickerView` 列表中间的数据即为用户选中的数据，并且此时，系统也会调用 `UIPickerView` 的如下代理方法来通知开发者用户的选择。

```

-(void)pickerView:(UIPickerView *)pickerView didSelectRow:(NSInteger) row
inComponent:(NSInteger) component{
    NSLog(@"%lu, %lu", row, component);
}

```

`pickerView:didSelectRow:inComponent:`方法在调用时, `row` 参数和 `component` 参数会将用户选择的行和分区的信息传递给开发者。

2.14 通过 CALayer 对视图进行修饰

CALayer 已经不属于 UIKit 框架中的内容, 事实上, 任何一个 UIView 的子类中, 都包含一个 CALayer 的属性, Layer 层是视图中专门用来渲染 UI 的一个层级, 而 View 层除了 UI 的展现外, 还封装了与用户交互的相关功能, 并且 View 层的 UI 展现也是通过 Layer 层来渲染的, 因此, 在 iOS 开发中, 很多动画的效果都是通过 CALayer 来实现的, 这些后面的内容会专门讲解, 本节将通过操作 Layer 层的一些简单的属性来对基本系统控件的 UI 表现进行渲染。

2.14.1 创建圆角的控件

UIKit 中的大多数控件创建时的尺寸都是规则矩形的, 在实际项目中, 开发者可能会需要使用到圆角的控件, 以 UIButton 控件为例, 使用 Xcode 创建一个名为 CALayerTest 的工程, 在 ViewController.m 文件的 `viewDidLoad` 方法中添加如下代码来设置 UIButton 控件的圆角。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIButton * btn = [UIButton buttonWithType:UIButtonTypeCustom];
    btn.frame=CGRectMake(100, 100, 100, 100);
    btn.backgroundColor = [UIColor redColor];
    btn.layer.masksToBounds = YES;
    btn.layer.cornerRadius = 10;
    [self.view addSubview:btn];
}
```

CALayer 对象的 `masksToBounds` 属性设置为 YES, 则对视图的边界进行修饰效果才会显现。 `cornerRadius` 属性设置圆角的半径, 如果控件的形状为矩形, 当这个值设置为控件边长的一半时, 控件的形状会变成圆形。运行工程, 效果如图 2-40 所示。

2.14.2 创建带边框的控件

在 iOS7 系统之前, 系统风格的 UIButton 控件是支持一种带边框的风格, 在 iOS7 之后, 系统不再支持创建出带边框的 UIButton 控件了, 但是开发者可以根据需要在 Layer 层做相关修饰来使 UIButton 控件带边框。

使用如下代码来创建带边框的 UIButton 控件:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIButton * btn = [UIButton buttonWithType:UIButtonTypeCustom];
```



```

btn.frame=CGRectMake(100, 100, 100, 100);
btn.backgroundColor = [UIColor redColor];
btn.layer.borderColor = [UIColor greenColor].CGColor;
btn.layer.borderWidth = 5;
[self.view addSubview:btn];
}

```

`borderColor` 属性设置边框的颜色, 这个属性需要设置为一个 `CGColor` 类型的对象, `UIColor` 对象可以通过调用 `CGColor` 方法来转换成 `CGColor` 对象。`borderWidth` 属性设置边框的宽度。运行工程, 效果如图 2-41 所示。

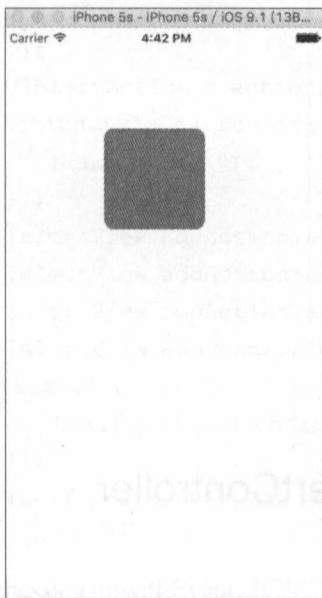


图 2-40 圆角控件

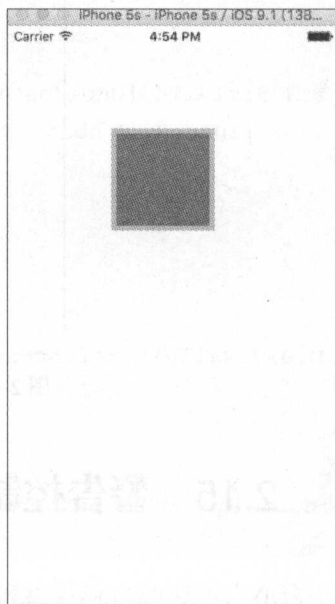


图 2-41 带边框的控件

2.14.3 为控件添加阴影效果

通过 `CALayer` 的属性, 还可以为控件添加一个立体的阴影效果, 使控件的展示有一定的 3D 视觉效果, 使用如下代码来为控件添加阴影:

```

- (void)viewDidLoad {
    [super viewDidLoad];
    UIButton * btn = [UIButton buttonWithType: UIButtonTypeCustom];
    btn.frame=CGRectMake(100, 100, 100, 100);
    btn.backgroundColor = [UIColor redColor];
    btn.layer.shadowColor = [UIColor grayColor].CGColor;
    btn.layer.shadowOffset = CGSizeMake(10, 10);
    btn.layer.shadowOpacity = 1;
    [self.view addSubview:btn];
}

```


`shadowColor` 属性设置阴影的颜色，必须为 `CGColor` 对象。`shadowOffset` 属性设置阴影的位置与原控件位置间的相对偏移，`shadowOpacity` 属性设置阴影的透明度，如果不设置，则默认为全透明，在界面上将看不到任何效果。运行工程，效果如图 2-42 所示。

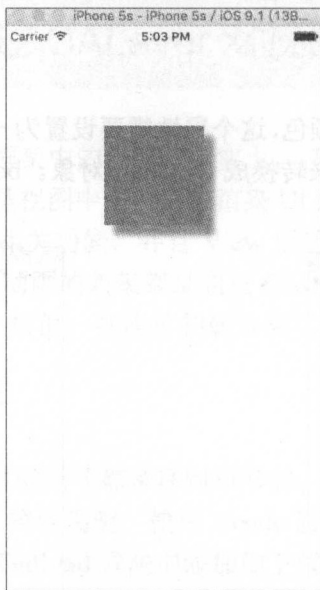


图 2-42 带阴影效果的控件

2.15 警告控制器——UIAlertController

在 iOS8 之前，UIKit 框架中有两个独立的视图控件用于在界面上弹出一个警告视图，分别是警告框控件 `UIAlertView` 和活动列表控件 `UIActionSheet`。iOS8 之后，已将 `UIAlertView` 和 `UIActionSheet` 进行了规范与统一，将这两个控件合并成新的视图控制器 `UIAlertController`。关于 `UIAlertView` 和 `UIActionSheet` 的使用在本章的扩展节中会有介绍，`UIAlertController` 相比于上面两个控件有很大的优势，一是方法进行了统一，便于开发者使用；二是 `UIAlertController` 提供了更强的扩展性接口；三是关于回调方法的处理。`UIAlertController` 使用 `block` 的形式代替了原先 `UIAlertView` 和 `UIActionSheet` 的代理方法，变得更加直观和简洁。

2.15.1 UIAlertController 的警告框

在项目中，经常会使用警告框来对用户的某些敏感操作提出警告提示。例如，当用户单击某个删除动作的按键时，一般会弹出一个警告框，警告用户是否确定删除，当用户单击删除后，才真正的执行删除操作。

使用 Xcode 创建一个名为 `UIAlertController_AlertView` 的工程，由于警告视图一般会在用户做某个操作之后弹出，因此我们可以在 `ViewController.m` 文件中实现一个当用户单击屏幕时会触发的方法来做演示，代码如下：

```

-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{
    UIAlertController * alertView = [UIAlertController alertControllerWithTitle:@"标题" message:@"警告的内容" preferredStyle:UIAlertControllerStyleAlert];

    UIAlertAction * action = [UIAlertAction actionWithTitle:@"按钮" style:UIAlertActionStyleDefault handler:^(UIAlertAction * _Nonnull action) {
        NSLog(@"click");
    }];

    UIAlertAction * action2 = [UIAlertAction actionWithTitle:@"取消" style:UIAlertActionStyleCancel handler:^(UIAlertAction * _Nonnull action) {
        NSLog(@"取消");
    }];

    UIAlertAction * action3 = [UIAlertAction actionWithTitle:@"注意" style:UIAlertActionStyleDestructive handler:^(UIAlertAction * _Nonnull action) {
        NSLog(@"注意");
    }];

    [alertView addAction:action];
    [alertView addAction:action2];
    [alertView addAction:action3];
    [alertView addTextFieldWithConfigurationHandler:^(UITextField * _Nonnull textField) {
        textField.placeholder=@"place";
    }];

    [self presentViewController:alertView animated:YES completion:nil];
}

```

`touchesBegan:withEvent:`方法在用户单击屏幕的时候被系统自动调用，在这个方法中实现对警告视图的创建和设置。

`alertControllerWithTitle:message:preferredStyle:`方法用于创建 `UIAlertController` 对象，第 1 个参数为警告视图的标题，第 2 个参数为警告视图的内容，第 3 个参数为警告视图的风格，有两种风格可以选择，分别是警告框风格 `UIAlertControllerStyleAlert` 和活动列表风格 `UIAlertControllerStyleActionSheet`。

`UIAlertAction` 可以理解为是一个封装了触发方法的选项按钮，其 `actionWithTitle:style:handler:`方法来创建一个 `UIAlertAction` 的对象，第 1 个参数为按钮的标题，第 2 个参数为按钮的风格，有 3 种风格可以选择，分别是默认风格 `UIAlertActionStyleDefault`，取消风格 `UIAlertActionStyleCancel` 和消极风格 `UIAlertActionStyleDestructive`。

`UIAlertController` 的 `addAction:`方法将向警告视图内添加一个选项按钮。`addTextFieldWithConfigurationHandler:`方法将向警告框中添加一个输入框，开发者可以在其中的 block 中对输入框 `TextField` 进行一些设置，运行工程，效果如图 2-43 与图 2-44 所示。



只有当警告控制器的风格为 `UIAlertControllerStyleAlert` 时才可以使用 `addTextFieldWithConfigurationHandler:`方法进行输入框的添加，否则会出现错误。

因为 `UIAlertController` 是一种视图控制器，因此要使用 `presentViewController:animated:completion:` 方法进行跳转。

对于 `UIAlertControllerStyleAlert` 风格而言，当选项按钮不超过两个时，按钮会横向并列排列，超过两个则会纵向排列。

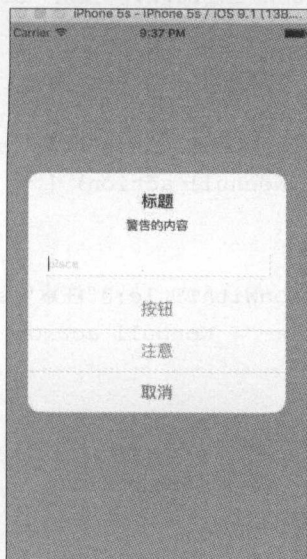


图 2-43 选项按钮超过两个的警告框

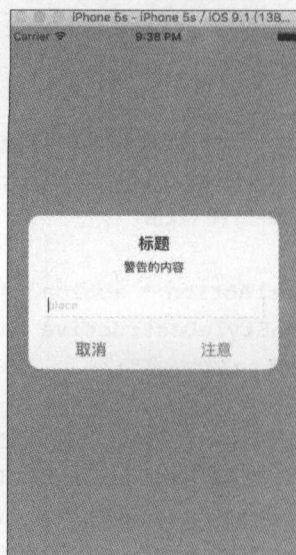


图 2-44 选项按钮为两个的警告框

2.15.2 UIAlertController 之活动列表

活动列表的作用与警告框类似，只是 UI 展现形式不同，使用 Xcode 创建一个名为 `UIAlertController_ActionSheet` 的工程，在 `ViewController.m` 文件中添加如下代码：

```
-(void)touchesBegan:(NSSet<UITouch * > *)touches withEvent:(UIEvent *)event {
    UIAlertController * actionSheet = [UIAlertController alertControllerWithTitle:@"标题" message:@"内容" preferredStyle:UIAlertControllerStyleActionSheet];

    UIAlertAction * action = [UIAlertAction actionWithTitle:@"one" style:UIAlertActionStyleDestructive handler:^(UIAlertAction * _Nonnull action) {
        NSLog(@"one");
    }];

    UIAlertAction * action2 = [UIAlertAction actionWithTitle:@"two" style:UIAlertActionStyleDefault handler:^(UIAlertAction * _Nonnull action) {
        NSLog(@"two");
    }];

    UIAlertAction * action3 = [UIAlertAction actionWithTitle:@"three" style:UIAlertActionStyleCancel handler:^(UIAlertAction * _Nonnull action) {
        NSLog(@"three");
    }];
}
```



```

    }];
    [actionSheet addAction:action];
    [actionSheet addAction:action2];
    [actionSheet addAction:action3];
    [self presentViewController:actionSheet animated:YES completion:nil];
}

```

运行工程，会看到如图 2-45 所示的效果。

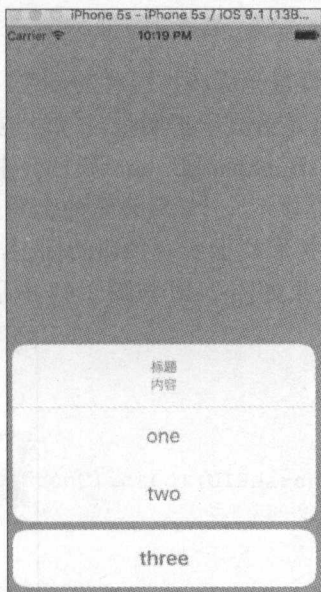


图 2-45 活动列表

2.16 扩展篇

本节将介绍尚未提到的一些在 iOS 开发中至关重要的 UI 控件，这些控件有从基本控件扩展而来的 `UISearchBar` 和 `UIDatePicker`，也有已经被弃用的 `UIAlertView` 和 `UIActionSheet`。

2.16.1 搜索栏控件——`UISearchBar`

`UISearchBar` 是 `UITextField` 与 `UISegmentedControl` 的组合与扩展，`UISearchBar` 的应用情景更加专一，专门用来创建搜索栏。使用 Xcode 创建一个名为 `UISearchBarTest` 的工程，在 `ViewController.m` 文件的 `viewDidLoad` 方法中添加如下代码：

```

- (void)viewDidLoad {
    [super viewDidLoad];
    UISearchBar * searchBar = [[UISearchBar alloc] initWithFrame:CGRectMake(
20, 100, 280, 30)];
}

```



```

searchBar.tintColor = [UIColor redColor];
searchBar.placeholder = @"请输入搜索内容";
searchBar.showsScopeBar = YES;
searchBar.showsCancelButton = YES;
searchBar.showsBookmarkButton=YES;
//searchBar.showsSearchResultsButton =YES;
[searchBar setScopeButtonTitles:@[@"one",@"two",@"three"]];
[self.view addSubview:searchBar];
}

```

UISearchBar 的 `tintColor` 属性设置光标和扩展栏的颜色, `placeholder` 属性设置搜索栏中的提示文字, `showsScopeBar` 属性设置是否显示扩展栏, 扩展栏实际上是一个 `UISegmentedControl` 控件。 `showsCancelButton` 和 `showsBookmarkButton` 属性分别设置是否显示取消按钮和图书按钮, 这里需要注意, 这两个按钮只能显示一个, 后设置的会覆盖先设置的。 `setScopeButtonTitles:` 方法需要传入一个数组, 用于设置扩展栏上所有按钮的标题, 传入的数组中元素的个数对应扩展栏上按钮的个数。运行工程, 效果如图 2-46 与图 2-47 所示。

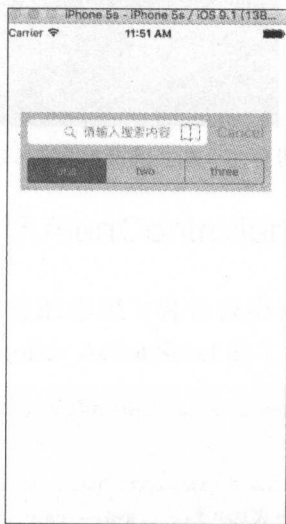


图 2-46 显示图书按钮的搜索栏

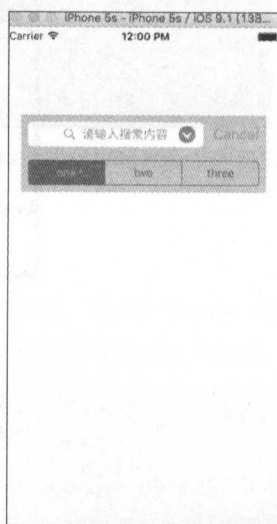


图 2-47 显示搜索结果按钮的搜索栏

UISearchBar 中单击按钮的触发逻辑和用户输入要搜索的字符时的相关监听都是通过 `UISearchBarDelegate` 协议中的方法来回调, 常用的代理方法的意义如下所示。

```

//单击切换扩展栏上按钮时触发的方法
-(void) searchBar:(UISearchBar *) searchBar selectedScopeButtonIndexDidChange: (NSInteger) selectedScope{

}

//搜索框中字符将要改变时触发的方法
-(BOOL) searchBar:(UISearchBar *) searchBar shouldChangeTextInRange: (NSRange) range replacementText: (NSString *) text{

```

```

        return YES;
    }
    //搜索框中字符已经改变后触发的方法
    -(void)searchBar:(UISearchBar *)searchBar textDidChange:(NSString *)searchText{

    }
    //单击图书按钮触发的方法
    -(void)searchBarBookmarkButtonClicked:(UISearchBar *)searchBar{

    }
    //单击取消按钮触发的方法
    -(void)searchBarCancelButtonClicked:(UISearchBar *)searchBar{

    }
    //单击搜索结果按钮触发的方法
    -(void)searchBarResultsListButtonClicked:(UISearchBar *)searchBar{

    }
    //单击键盘上的搜索键触发的方法
    -(void)searchBarSearchButtonClicked:(UISearchBar *)searchBar{

    }
    //搜索栏将要开始编辑时触发的方法
    -(BOOL)searchBarShouldBeginEditing:(UISearchBar *)searchBar{
        return YES;
    }
    //搜索栏将要结束编辑时触发的方法
    -(BOOL)searchBarShouldEndEditing:(UISearchBar *)searchBar{
        return YES;
    }
    //搜索栏已经开始编辑时触发的方法
    -(void)searchBarTextDidBeginEditing:(UISearchBar *)searchBar{

    }
    //搜索栏已经结束编辑时触发的方法
    -(void)searchBarTextDidEndEditing:(UISearchBar *)searchBar{

    }
}

```

2.16.2 日期时间选择器——UIDatePicker

UIDatePicker 在 UI 展现方面和 UIPickerView 十分相似，但是 UIDatePicker 并非是继承于

UIPickerView 的子类，它是继承于 UIControl 的一个独立控件，因此，在实现逻辑上，UIDatePicker 不是采用代理方法的模式。

使用 Xcode 创建一个名为 UIDatePickerTest 的工程，在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIDatePicker * datePicker = [[UIDatePicker alloc] initWithFrame:CGRectMake(20, 100, 280, 150)];
    datePicker.datePickerMode = UIDatePickerModeTime;
    [datePicker addTarget:self action:@selector(change:) forControlEvents:
    UIControlEventValueChanged];
    [self.view addSubview:datePicker];
}
```

UIDatePicker 控件的 datePickerMode 属性设置控件的风格，可选的枚举意义如下：

```
typedef NS_ENUM(NSInteger, UIDatePickerMode) {
    UIDatePickerModeTime,           // 时间模式
    UIDatePickerModeDate,           // 日期模式
    UIDatePickerModeDateAndTime,    // 日期和时间模式
    UIDatePickerModeCountDownTimer, // 计时模式
};
```

各种风格模式的效果如图 2-48~图 2-51 所示。UIDatePicker 控件使用 addTarget:action:forControlEvents:方法来添加触发事件，在触发函数中可以获取到 UIDatePicker 中当前的日期时间信息，如下所示：

```
-(void)change:(UIDatePicker *)picker{
    NSLog(@"%@", picker.date);
}
```

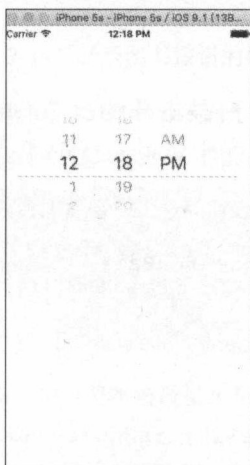


图 2-48 UIDatePickerModeTime

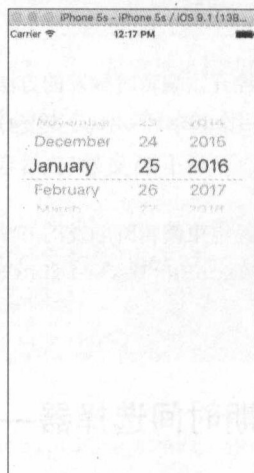


图 2-49 UIDatePickerModeDate

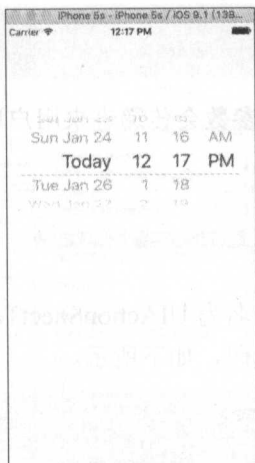


图 2-50 UIDatePickerModeDateAndTime

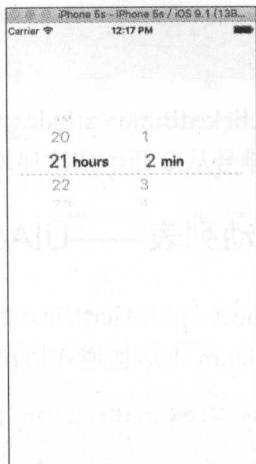


图 2-51 UIDatePickerModeCountDownTimer

2.16.3 警告视图——UIAlertView

前面提到过，在 iOS8 之后，系统采用统一的 UIAlertController 代替了 UIAlertView 和 UIActionSheet，但是在目前所有 iOS 各系统用户的占比中，iOS7 系统仍然存在。既然存在，开发者在编写代码时就要考虑到对 iOS7 系统的兼容性，如果在 iOS7 系统上运行使用 UIAlertController 的方法的程序，程序将会直接崩溃。

使用 Xcode 创建一个名为 UIAlertViewTest 的工程，在 ViewController.m 文件中添加如下函数：

```
-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{
    UIAlertView * alert = [[UIAlertView alloc] initWithTitle:@"标题" message:
@"内容" delegate:self cancelButtonTitle:@"取消" otherButtonTitles:@"确定", nil];
    [alert show];
}
```

当用户单击屏幕时，touchesBegan:withEvent:方法会被调用，UIAlertView 的 initWithTitle:message:delegate:cancelButtonTitle:otherButtonTitles:方法创建 UIAlertView 的对象，这个方法中第 1 个参数为警告框的标题，第 2 个参数为警告框的内容，第 3 个参数设置代理，第 4 个参数设置取消按钮的标题，第 5 个参数设置其他按钮的标题，在第 5 个参数之后，可以继续添加标题参数，以逗号进行分隔。调用 show 方法来对警告框进行展现。

UIAlertView 的按钮触发方法是通过代理来回调的，首先需要在 ViewController.m 文件中遵守的协议如下：

```
@interface ViewController ()<UIAlertViewDelegate>

@end
```

实现下面的代理方法来监听用户的单击按钮操作：

```
-(void>alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)
buttonIndex{
```



```

        NSLog(@"click");
    }

```

alertView:clickedButtonAtIndex:代理方法中的 buttonIndex 参数会传递进来用户单击的按钮标号, 按钮的排号从 0 开始依次递增。

2.16.4 活动列表——UIActionSheet

UIActionSheet 与 UIAlertView 的用法十分相似, 创建一个名为 UIActionSheetTest 的工程, 在 ViewController.m 中添加遵守协议和创建活动列表的相关代码, 如下所示。

```

@interface ViewController () <UIActionSheetDelegate>

@end

-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{
    UIActionSheet * actionSheet = [[UIActionSheet alloc] initWithTitle:@"标题" delegate:self cancelButtonTitle:@"取消" destructiveButtonTitle:@"删除" otherButtonTitles:@"确定", nil];
    [actionSheet showInView:self.view];
}

-(void)actionSheet:(UIActionSheet *)actionSheet clickedButtonAtIndex:(NSInteger)buttonIndex{
    NSLog(@"click");
}

```

其中需要注意的地方在于 UIActionSheet 展现时调用 showInView:方法。



提示

iOS 系统是向下兼容的, 例如在 iOS 9 系统中使用被弃用的 iOS8 之前的方法, Xcode 会给出警告, 而程序依然可以很好地工作, 但是如果在低 iOS 版本中使用了高版本才有的方法, 程序会直接崩溃。

2.17 实战：登录注册界面的搭建

本节将实现一个简易登录注册的界面, 目的是练习综合使用本章中介绍的这些独立的 UI 控件, 所有复杂控件都是由简单控件组合与扩展而来的, 所有复杂的界面也都是将独立的控件进行组合和串联使用的。

使用 Xcode 创建一个名为 LoginView 的工程。分析一下需求, 我们需要两个界面, 一个作为登录界面, 一个作为注册界面。将框架中创建好的 ViewController 作为登录界面, 再新建一个文件来作为注册界面。在 Xcode 的文件导航区单击右键, 选择 NewFile, 如图 2-52 所示。

在弹出的窗口中选择 Cocoa Touch Class 选项, 再单击 Next, 如图 2-53 所示。

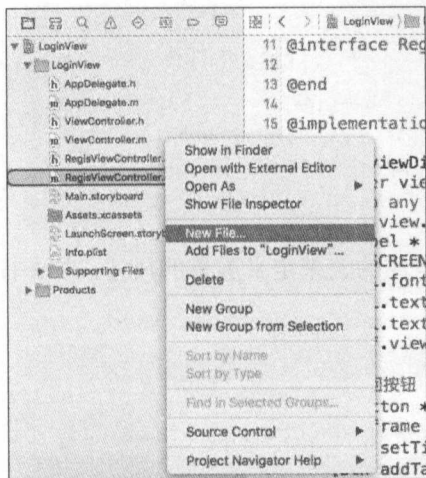


图 2-50 新建一个文件

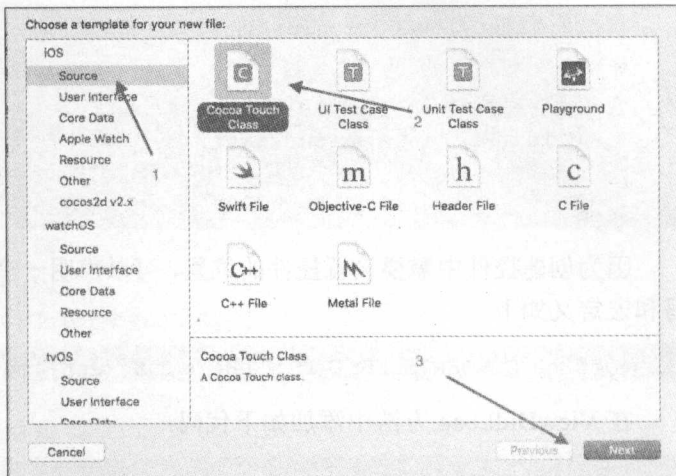


图 2-52 创建 iOS 类文件

在弹出的选项窗口中继承的父类一栏选择 `UIViewController`，类名取为 `RegisController`，如图 2-54 所示，然后单击 `Next`。在弹出的选择创建路径的窗口中直接单击 `Create`，这时会看到 Xcode 的文件导航栏中多了两个文件。

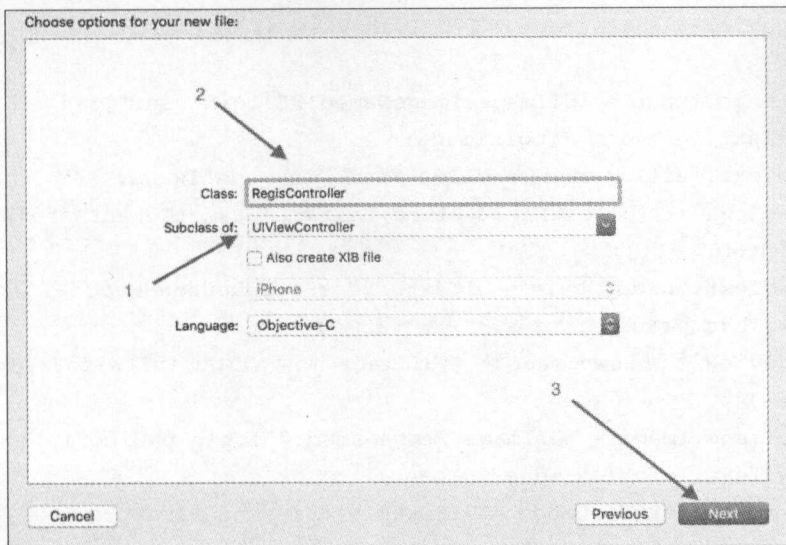
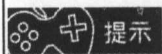


图 2-54 设置创建类的相关参数



提示

在 iOS 开发中，有一些规则是开发者们心照不宣的，命名规则就是这样，虽然语法中没有严格的规定，但一般在对变量和对象取名时会采用首字母小写的驼峰命名法，例如 `oneTitle`，而类名的首字母一般要大写，如 `OneClass`。

创建了注册界面的文件，再回到 `ViewController.m` 文件中，先来编写登录界面的代码，这里将用户名输入框和密码输入框声明成员变量，以便于在其他函数中使用这些对象，代码如下所示。

```

@interface ViewController ()
{
    UITextField * _loginText;
    UITextField * _passwdText;
}
@end

```

因为创建控件中需要设置控件的位置，可以声明一个全局的宏来定义屏幕的尺寸，方法书写和宏定义如下：

```
#define SCREEN_SIZE [UIScreen mainScreen].bounds.size
```

在 ViewDidLoad 方法中添加如下代码：

```

- (void)viewDidLoad {
    //创建用户名和密码框
    _loginText = [[UITextField alloc] initWithFrame:CGRectMake(20, 80, SCREEN_SIZE.width-40, 30)];
    _loginText.borderStyle = UITextBorderStyleRoundedRect;
    _loginText.placeholder = @"请输入用户名";
    UIImageView * loginImage = [[UIImageView alloc] initWithFrame:CGRectMake(0, 0, 20, 20)];
    loginImage.image = [UIImage imageNamed:@"login_user"];
    _loginText.leftView = loginImage;
    _loginText.leftViewMode = UITextFieldViewModeAlways;
    _passwdText = [[UITextField alloc] initWithFrame:CGRectMake(20, 130, SCREEN_SIZE.width-40, 30)];
    _passwdText.borderStyle = UITextBorderStyleRoundedRect;
    _passwdText.placeholder = @"请输入密码";
    UIImageView * passwdImage = [[UIImageView alloc] initWithFrame:CGRectMake(0, 0, 20, 20)];
    passwdImage.image = [UIImage imageNamed:@"login_pwdico"];
    _passwdText.leftView = passwdImage;
    _passwdText.leftViewMode = UITextFieldViewModeAlways;
    [self.view addSubview:_loginText];
    [self.view addSubview:_passwdText];

    //创建登录按钮和注册按钮
    UIButton * btn = [UIButton buttonWithType:UIButtonTypeSystem];
    btn.frame = CGRectMake(SCREEN_SIZE.width/4-50, 180, 100, 30);
    [btn setTitle:@"登录" forState:UIControlStateNormal];
    btn.layer.masksToBounds=YES;
    btn.layer.cornerRadius = 10;
    btn.backgroundColor = [UIColor cyanColor];
}

```



```

[btn addTarget:self action:@selector(login) forControlEvents:UIControlEventTouchUpInside];

UIButton * btn2 = [UIButton buttonWithType:UIButtonTypeSystem];
btn2.frame = CGRectMake(SCREEN_SIZE.width/4*3-50, 180, 100, 30);
[btn2 setTitle:@"注册" forState:UIControlStateNormal];
btn2.layer.masksToBounds=YES;
btn2.layer.cornerRadius = 10;
btn2.backgroundColor = [UIColor cyanColor];
[btn2 addTarget:self action:@selector(regis) forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:btn];
[self.view addSubview:btn2];
}

```

上面代码创建了两个输入框和两个按钮，两个按钮的触发方法实现代码如下所示：

```

-(void)regis{
    RegisViewController * con = [[RegisViewController alloc] init];
    [self presentViewController:con animated:YES completion:nil];
}

-(void)login{
    if (_loginText.text.length==0) {
        UIAlertController * alertCon = [UIAlertController alertControllerWithTitle:@"温馨提示" message:@"请输入用户名" preferredStyle:UIAlertControllerStyleAlert];

        UIAlertAction * action = [UIAlertAction actionWithTitle:@"好的" style:UIAlertActionStyleDefault handler:^(UIAlertAction * _Nonnull action) {

        }];
        [alertCon addAction:action];
        [self presentViewController:alertCon animated:YES completion:nil];
        return;
    }
    if (_passwdText.text.length==0) {
        UIAlertController * alertCon = [UIAlertController alertControllerWithTitle:@"温馨提示" message:@"请输入密码" preferredStyle:UIAlertControllerStyleAlert];

        UIAlertAction * action = [UIAlertAction actionWithTitle:@"好的" style:UIAlertActionStyleDefault handler:^(UIAlertAction * _Nonnull action) {

        }];
        [alertCon addAction:action];
    }
}

```



```

        [self presentViewController:alertCon animated:YES completion:nil];
        return;
    }

    UIAlertController * alertCon = [UIAlertController alertControllerWithTitle:@"温馨提示" message:@"登录成功" preferredStyle:UIAlertControllerStyleAlert];
    UIAlertAction * action = [UIAlertAction actionWithTitle:@"好的" style:UIAlertActionStyleDefault handler:^(UIAlertAction * _Nonnull action) {

    }];
    [alertCon addAction:action];
    [self presentViewController:alertCon animated:YES completion:nil];
}

```

在注册方法 `regis` 中，创建了注册界面并进行跳转。在登录按钮的触发方法 `login` 中先进行了用户名框是否为空的判断，如果为空，则会弹出警告框提示用户，之后进行了密码框是否为空的判断，如果为空，弹出警告框提示用户，如果用户名框和密码框都不为空，则弹出登录成功的提示。



提示

在一个类中引用到另一个类的时候，要引入被引用的类的头文件，例如上面使用了注册界面类，需要添加如下代码：

```
#import "RegisViewController.h"
```

在注册界面 `RigisController.m` 文件中添加一个返回按钮和其触发方法，代码如下：

```

- (void)viewDidLoad {
    [super viewDidLoad];
    self.view.backgroundColor = [UIColor whiteColor];
    UILabel * label = [[UILabel alloc] initWithFrame:CGRectMake(20, 100, SCREEN_SIZE.width-40, 60)];
    label.font = [UIFont systemFontOfSize:23];
    label.text = @"注册界面";
    label.textAlignment = NSTextAlignmentCenter;
    [self.view addSubview:label];

    //返回按钮
    UIButton * btn = [UIButton buttonWithType:UIButtonTypeSystem];
    btn.frame = CGRectMake(SCREEN_SIZE.width/2-50, 220, 100, 30);
    [btn setTitle:@"返回" forState:UIControlStateNormal];
    [btn addTarget:self action:@selector(retu) forControlEvents:UIControlEventTouchUpInside];
    [self.view addSubview:btn];
}

```

```
-(void)retu{  
    [self dismissViewControllerAnimated:YES completion:nil];  
}
```

一个简易的登录注册界面就搭建完成了，效果如图 2-55 所示。

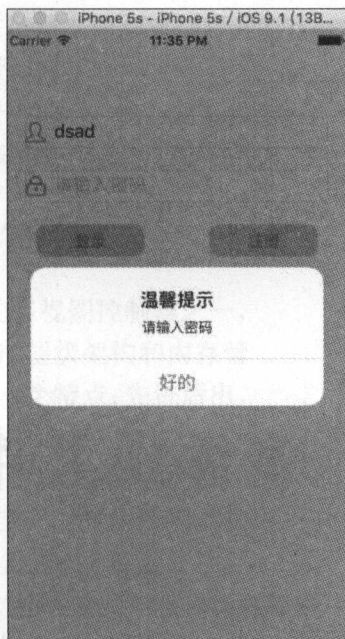


图 2-55 登录注册界面

第 3 章

高级 UI 控件

在第 2 章中介绍的大多是一些独立的 UI 控件，除了 `UIViewController` 和 `UIAlertController` 属于 Controller 层，其他的控件都属于 View 层。在 MVC 设计模式中，Controller 层是核心，控制器将视图和数据进行组合，并处理交互逻辑。在 iOS 的 UIKit 框架中，`UIViewController` 是最基本的视图控制器，除此之外，还有一些更加高级的视图控制器来完成更加复杂的 UI 层级结构。其中最典型的两种结构化控制器便是导航控制器 `UINavigationController` 和标签控制器 `UITabBarController`。本章将着重对这两种控制器进行介绍。

在 View 层，也存在一些控件并非像第 2 章中介绍的独立控件那样简单，复杂控件中最典型也最常用的是表格视图 `UITableView`，它需要通过数据源和代理两个协议来进行 UI 设置，通过 `UITableViewCell` 来进行 UI 展现，比之更加复杂的是 `UICollectionView`，而它们的核心滚动功能又都是通过 `UIScrollView` 来实现的。因此，这 3 种视图也是本章将要介绍的重点。

通过本章的学习，读者能够掌握：

1. 使用导航控制器设计堆栈层级结构的项目框架
2. 使用标签控制器设计并列结构的项目框架
3. 综合导航控制器与标签控制器设计项目框架
4. 滚动视图 `UIScrollView` 的使用
5. 表格视图 `UITableView` 和集合视图 `UICollectionView` 的使用
6. 学会使用网页视图展现网页内容

3.1 导航控制器——UINavigationController

严格来说，导航控制器并不是视图控制器，其内部并没有视图层，可以理解为导航控制器是视图控制器的控制器，导航控制器是专门来管理 `ViewController` 的一个容器类，导航控制器内如果没有 `ViewController` 或者只有一个 `ViewController`，那它将没有什么用武之地，并且，通过导航控制器管理视图控制器的应用会以层级的结构进行框架搭建。

3.1.1 导航控制器的工作原理

导航控制器是 iOS 中常用的多视图控制器之一，因其采用堆栈的设计模式，在视图层级架构和内存管理上优势明显。堆栈模式有这样一个特点：先进后出，后进先出，图 3-1 可以很好地表示这种结构。

如图 3-1 所示，绿色的方块代表视图控制器，红色的容器代表导航控制器。当一个视图控制器被 `push`（压入栈操作）进导航控制器中时，它会展现在当前的屏幕界面上，如果再有一个视图控制器被 `push` 进导航容器，则后入的会覆盖替换先前的视图控制器展示在屏幕上，对于 `pop` 操作（弹出栈操作）来说，后被压入导航容器的视图控制器会被先弹出，例如图 3-1 中的视图控制器 3 被弹出后，屏幕界面上将展现视图控制器 2，视图控制器 2 再被弹出后，屏幕界面会展现视图控制器 1。导航控制器就是通过这样的层级结构来管理各个视图控制器的。在实际应用中，应用程序通常会以如图 3-2 所示的方法运用导航控制器。

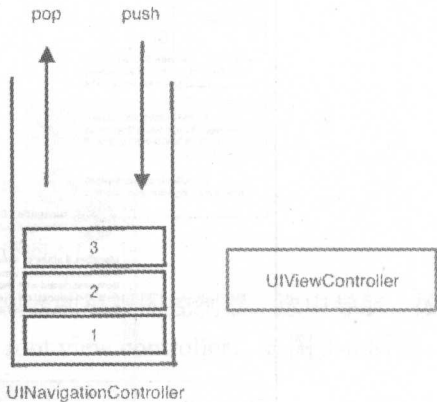


图 3-1 导航控制器的堆栈结构

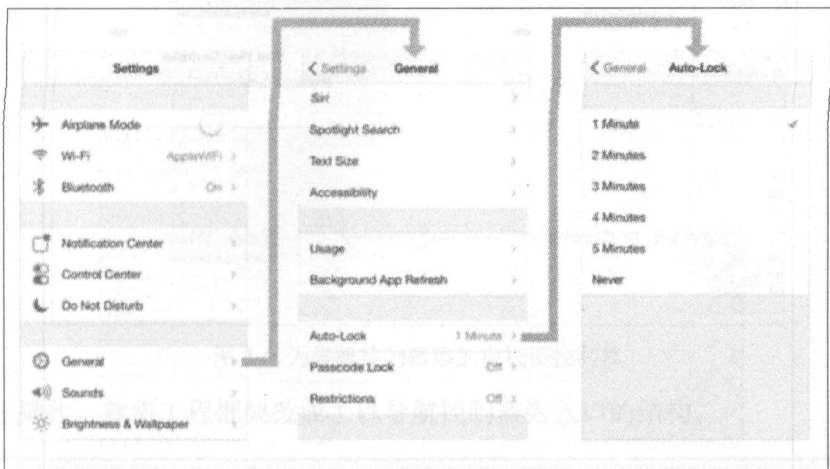


图 3-2 实际应用中的导航控制器模型

3.1.2 使用导航控制器进行多界面搭建

使用 Xcode 开发工具创建一个名为 UINavigationControllerTest 的工程, Xcode 创建的工程模板默认是以 UIViewController 为根视图控制器的, 需要将其修改为导航控制器, 单击 Main.storyboard 文件, 在 Xcode 右下方的控件窗口中选中 Navigation Controller 控件, 如图 3-3 所示。

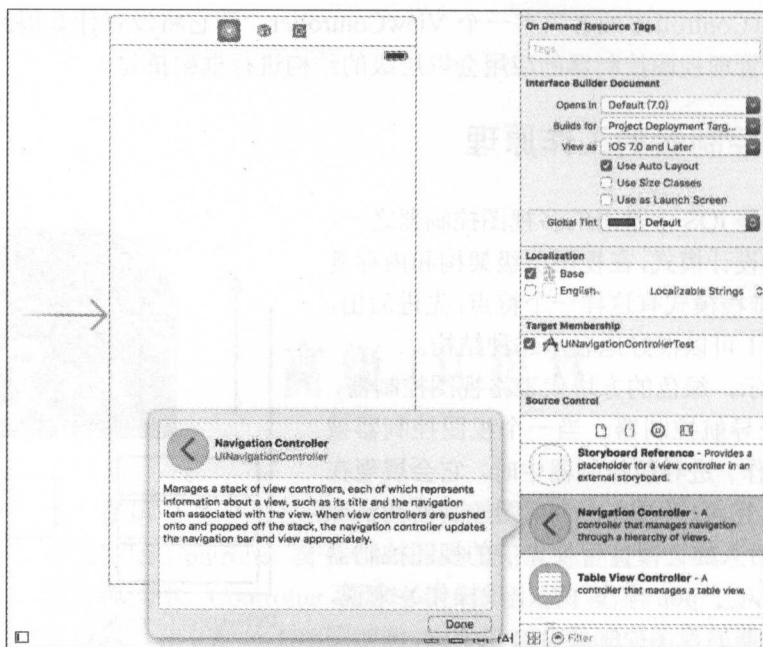


图 3-3 storyboard 中的导航控制器控件

将选中的导航控制器拖入编辑窗口中, 这时导航控制器将自带一个 rootViewController, 即导航的根视图控制器, 如图 3-4 所示。

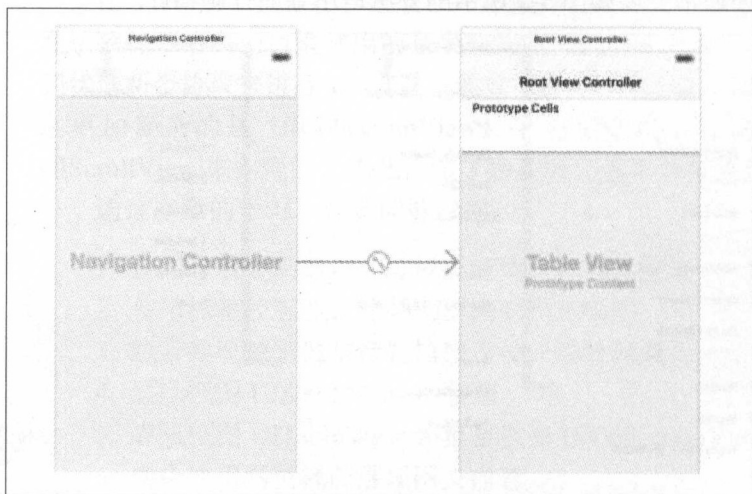


图 3-4 向 storyboard 中添加导航控制器

将导航控制器自带的 rootViewController 删掉，并将导航控制器设置为程序的入口，如图 3-5 所示。

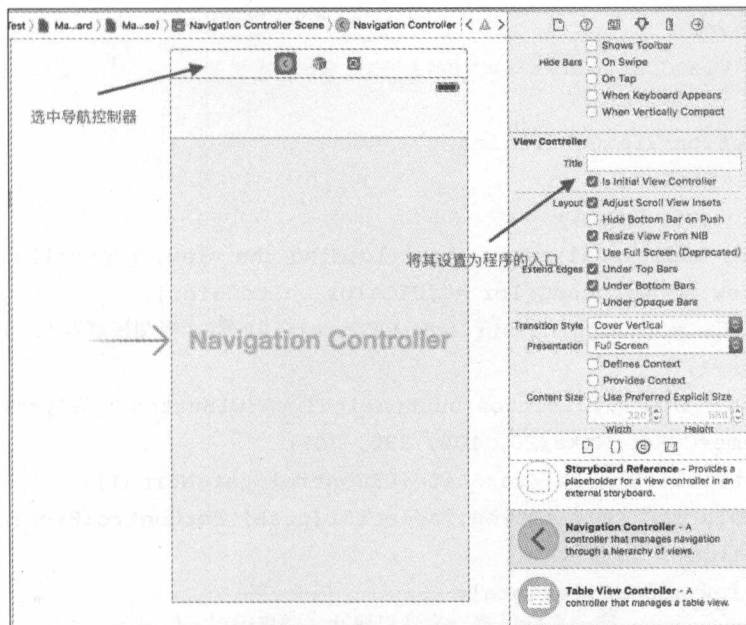


图 3-5 设置导航控制器为程序的入口

将工程模板中自带的 ViewController 设置为导航控制器的根视图控制器，选中导航，按住 control 不放，将鼠标拖动至 ViewController 中后，选择 root view controller，如图 3-6 所示。

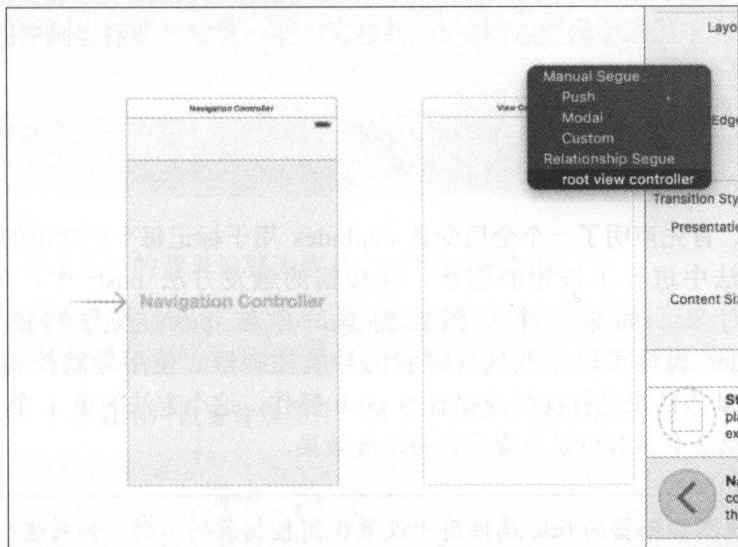


图 3-6 为导航控制器添加根视图控制器

完成如上操作，就将工程框架改成了以导航控制器为入口的结构。



提示

在 storyboard 文件的编辑界面，通过单击右键，可以设置屏幕的缩放比例。

进入 ViewController.m 文件，现在向其中添加使用导航控制器进行视图控制器跳转的相关代码，这里采用单击按钮的方式触发导航控制器的 push 操作，代码如下：

```
int conIndex=1;
@interface ViewController ()
@end
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    self.view.backgroundColor = [UIColor whiteColor];
    self.title = [NSString stringWithFormat:@"第%d 视图控制器", conIndex];
    conIndex++;
    UIButton * btn = [UIButton buttonWithType:UIButtonTypeSystem];
    btn.frame=CGRectMake(20, 100, 280, 30);
    [btn setTitle:@"push" forState:UIControlStateNormal];
    [btn addTarget:self action:@selector(push) forControlEvents:UIControlEventTouchUpInside];
    [self.view addSubview:btn];
}
-(void)push{
    ViewController * con = [[ViewController alloc] init];
    con.title = [NSString stringWithFormat:@"第%d 视图控制器", conIndex];
    [self.navigationController pushViewController:con animated:YES];
}
-(void)dealloc{
    conIndex--;
}
@end
```

上面代码中，首先声明了一个全局变量 conIndex 用于标记每个创建出的视图控制器，在 viewDidLoad 方法中进行了按钮的创建，在按钮的触发方法 push 中，创建了一个新的 ViewController 对象，如果一个视图控制器对象被 push 入导航控制器，则它的 navigationController 属性可以获取到管理它的导航控制器。使用导航控制器的 pushViewController:animated: 方法来进行视图控制器的 push 操作，这个方法第 1 个参数为要被 push 的视图控制器，第 2 个参数可以设置是否带动画效果。



提示

视图控制器的 title 属性用于设置视图控制器的名称，如果这个视图控制器在导航控制器的管理下，则这个名称会显示在导航栏上。

UITableViewController 的 dealloc 方法在其被释放时调用。

运行工程，单击 push 按钮，可以看到导航控制器进行视图控制器的切换效果，并且在新

push 出来的视图控制器中，导航栏的左侧会出现一个返回按钮，单击返回按钮会执行导航控制器的 pop 操作，如图 3-7 所示。

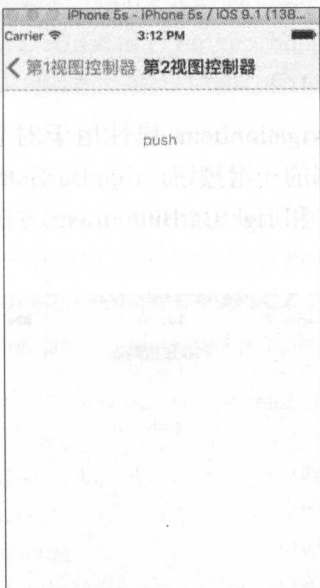


图 3-7 使用导航控制器进行多视图控制器管理

3.1.3 导航栏 UINavigationController

如图 3-7 所示，导航控制器内部提供一个 UINavigationController 类型的导航栏，导航栏用于显示当前栈顶的视图控制器标题和放置一些交互按钮。在视图控制器中通过下面代码可以设置导航栏的背景颜色。

```
self.navigationController.navigationBar.barTintColor = [UIColor purpleColor];
```

效果如图 3-8 所示。

同样，也可以将导航栏的背景设置为图片，使用如下代码：

```
[self.navigationController.navigationBar setBackgroundImage:[UIImage imageNamed:@"image"] forBarMetrics:UIBarMetricsDefault];
```

上面方法中第 1 个参数用户设置背景图片，第 2 个参数设置屏幕的状态，枚举意义如下：

```
typedef NS_ENUM(NSInteger, UIBarMetrics) {
    UIBarMetricsDefault, // 默认竖屏模式
    UIBarMetricsCompact, // 横屏模式
}
```

运行工程，效果如图 3-9 所示。

当一个视图控制器被 push 出来后，其导航栏中会自动带一个返回按钮，UINavigationController 也提供了接口供开发者自定义一些导航栏上的按钮，代码示例如下：


```
UIBarButtonItem * btnItem1 = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemAction target:self action:@selector(click)];
UIBarButtonItem * btnItem2 = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self action:@selector(click)];
self.navigationItem.leftBarButtonItems = @[btnItem1,btnItem2];
```

UIViewController 对象的 `navigationItem` 属性用于对导航栏上的按钮进行相关设置，`leftBarButtonItems` 设置导航栏左侧的一组按钮，`rightBarButtonItems` 设置导航栏右侧的一组按钮，当然，也有 `leftBarButtonItem` 和 `rightBarButtonItem` 方法用于设置导航栏左右侧的一个按钮。设置效果如图 3-10 所示。

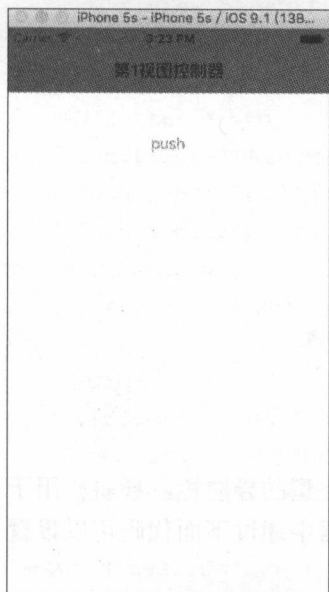


图 3-8 为导航栏添加背景颜色

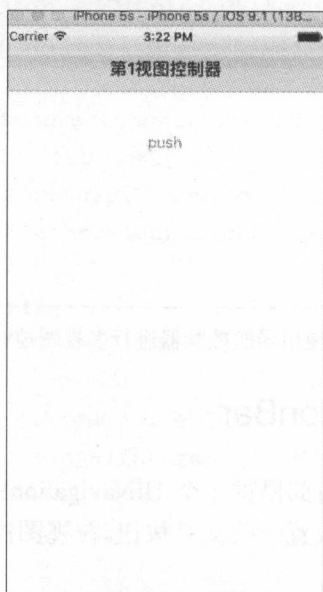


图 3-9 为导航栏设置背景图案

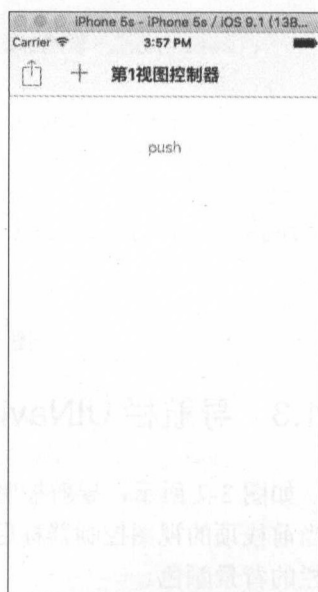


图 3-10 为导航栏添加自定义按钮



提示

为导航栏添加的左侧按钮并不自带 `pop` 方法，如果需要其自带 `pop` 方法，使用如下代码设置：

```
self.navigationItem.backBarButtonItem = btnItem1;
```

3.1.4 导航按钮 UIBarButtonItem

导航栏上的按钮需要设置为 `UIBarButtonItem` 的对象，`UIBarButtonItem` 除了可以使用系统准备好的一个 UI 风格外，还可以进行完全的自定义，初始化 `UIBarButtonItem` 的方法有如下几种：

```
UIBarButtonItem * item1= [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self action:@selector(click)];
UIBarButtonItem * item2 = [[UIBarButtonItem alloc] initWithCustomView:[UIView alloc]init];
```

```

UIBarButtonItem * item3 = [[UIBarButtonItem alloc] initWithImage:[UIImage
imageNamed:@"image"] style:UIBarButtonItemStylePlain target:self action:@sele
ctor(click)];

UIBarButtonItem * item4 = [[UIBarButtonItem alloc] initWithTitle:@"标题" st
yle:UIBarButtonItemStylePlain target:self action:@selector(click)];

```

在上面的代码中，item1 使用 initWithBarButtonSystemItem:target:action:方法来创建，这里会传入一个系统风格的枚举，后边会有介绍。item2 使用 initWithCustomView:方式来创建，这里可以传入一个自定义的 UIView 或者其子类对象。item3 使用 initWithImage:style:target:action:方法来创建，这个方法可以创建一个自定义图片的导航按钮。item4 使用 initWithTitle:style:target:action:方法来创建，这个方法创建一个自定义标题的导航按钮。

关于 initWithBarButtonSystemItem:target:action:方法，其中支持的按钮风格枚举意义如下所示：

```

typedef NS_ENUM(NSInteger, UIBarButtonSystemItem) {
    UIBarButtonSystemItemDone,           //完成风格的按钮
    UIBarButtonSystemItemCancel,         //取消风格的按钮
    UIBarButtonSystemItemEdit,           //编辑风格的按钮
    UIBarButtonSystemItemSave,           //保存风格的按钮
    UIBarButtonSystemItemAdd,            //添加风格的按钮
    UIBarButtonSystemItemFlexibleSpace,  //占位按钮 不显示
    UIBarButtonSystemItemFixedSpace,    //占位按钮 不显示
    UIBarButtonSystemItemCompose,        //构图风格的按钮
    UIBarButtonSystemItemReply,          //回复风格的按钮
    UIBarButtonSystemItemAction,         //功能风格的按钮
    UIBarButtonSystemItemOrganize,       //机构风格的按钮
    UIBarButtonSystemItemBookmarks,      //书签风格的按钮
    UIBarButtonSystemItemSearch,         //搜索风格的按钮
    UIBarButtonSystemItemRefresh,        //刷新风格的按钮
    UIBarButtonSystemItemStop,           //停止风格的按钮
    UIBarButtonSystemItemCamera,         //相机风格的按钮
    UIBarButtonSystemItemTrash,          //清除风格的按钮
    UIBarButtonSystemItemPlay,           //播放风格的按钮
    UIBarButtonSystemItemPause,          //暂停风格的按钮
    UIBarButtonSystemItemRewind,         //倒带风格的按钮
    UIBarButtonSystemItemFastForward,    //前进风格的按钮
    UIBarButtonSystemItemUndo,           //撤销风格的按钮
    UIBarButtonSystemItemRedo,           //重做风格的按钮
    UIBarButtonSystemItemPageCurl,       //页面卷曲风格按钮 只能用于工具栏
};

```

部分按钮显示效果如图 3-11~图 3-31 所示。

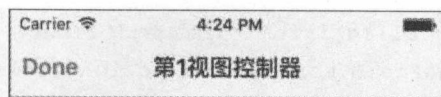


图 3-11 UIBarButtonItemSystemItemDone

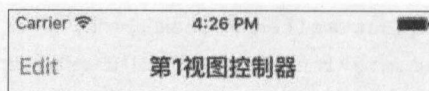


图 3-12 UIBarButtonItemSystemItemCancel

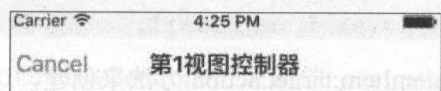


图 3-13 UIBarButtonItemSystemItemEdit

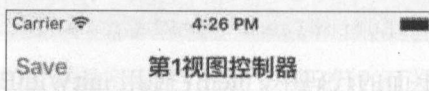


图 3-14 UIBarButtonItemSystemItemSave

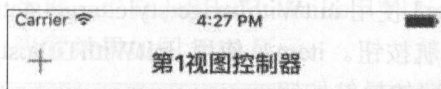


图 3-15 UIBarButtonItemSystemItemAdd

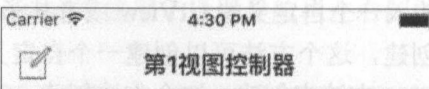


图 3-16 UIBarButtonItemSystemItemCompose

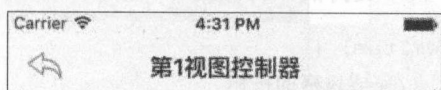


图 3-17 UIBarButtonItemSystemItemReply

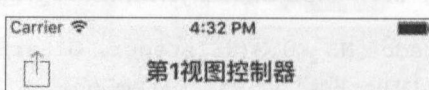


图 3-18 UIBarButtonItemSystemItemAction

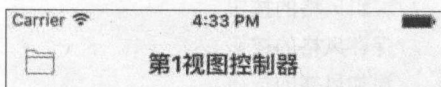


图 3-19 UIBarButtonItemSystemItemOrg

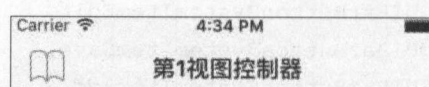


图 3-20 UIBarButtonItemSystemItemBookmarks

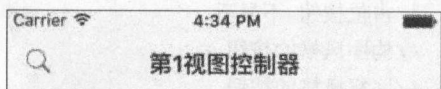


图 3-21 UIBarButtonItemSystemItemSearch

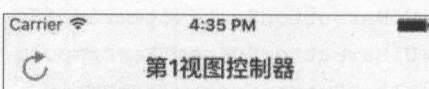


图 3-22 UIBarButtonItemSystemItemRefresh

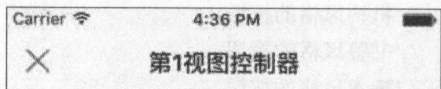


图 3-23 UIBarButtonItemSystemItemStop

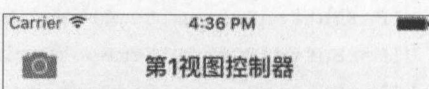


图 3-24 UIBarButtonItemSystemItemCamera

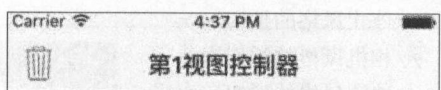


图 3-25 UIBarButtonItemSystemItemTrash

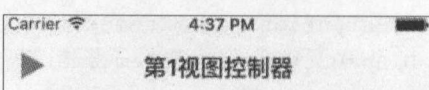


图 3-26 UIBarButtonItemSystemItemPlay

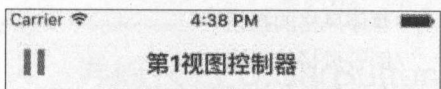


图 3-27 UIBarButtonItemSystemItemPause

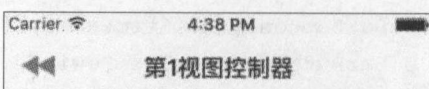


图 3-28 UIBarButtonItemSystemItemRewind

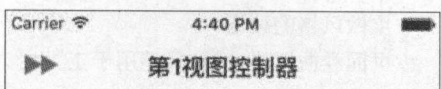


图 3-29 UIBarButtonItemSystemItemFastForward

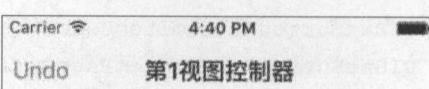


图 3-30 UIBarButtonItemSystemItemUndo

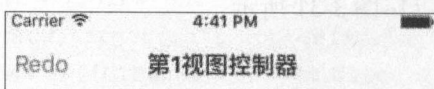


图 3-31 UIBarButtonItemSystemItemRedo

3.1.5 导航控制器的工具栏

导航控制器中除了自带一个导航栏 `UINavigationController` 之外，还带有一个工具栏 `UIToolBar` 控件，只是工具栏并不常用，导航控制器默认是将其隐藏的，可以使用如下代码来取消工具栏的隐藏。

```
self.navigationController.toolbarHidden=NO;
self.navigationController.toolbar.barTintColor = [UIColor redColor];
```

导航控制器的工具栏出现在视图的底部，运行工程，效果如图 3-32 所示。

工具栏中一般会放置一些常用的小工具按钮，示例如下：

```
UIBarButtonItem * btnItem1 = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemPlay target:self action:@selector(click)];
UIBarButtonItem * btnItem2 = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self action:@selector(click)];
self.toolbarItems = @[btnItem1,btnItem2];
```

工具栏中添加的按钮也是 `UIBarButtonItem` 类型的对象，运行工程，效果如图 3-33 所示。

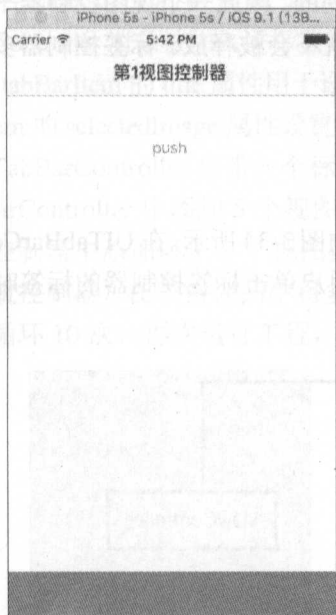


图 3-32 导航控制器中的工具栏

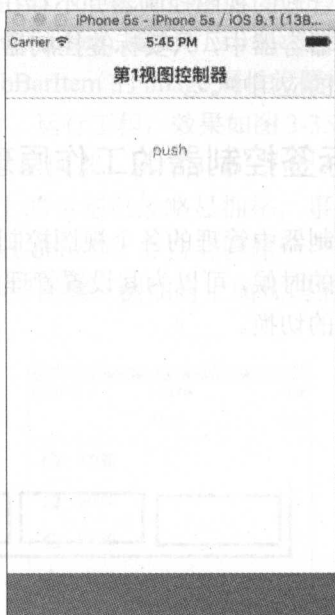


图 3-33 向工具栏中添加小工具按钮

3.1.6 iOS8 之后导航控制器的一些有趣功能

读者可能注意到过，一些用户体验十分优秀的软件对导航控制器都有这样的一些细微操作：网页浏览器在向上滑动浏览网页内容的时候，导航控制器的导航栏和工具栏会自动隐藏；读书软件在单击屏幕后，导航控制器的导航栏和工具栏会自动隐藏；视频播放器软件在横屏播放时导航控制器的导航栏和工具栏会自动隐藏。这些小细节的功能都能在很大程度上改善用户

的体验，在 iOS8 之前要做到这些，可能需要开发者手动添加许多逻辑代码，在 iOS8 之后，UINavigationController 为开发者提供了这些功能，代码如下所示：

```
self.navigationController.hidesBarsWhenVerticallyCompact=YES;
self.navigationController.hidesBarsOnTap=YES;
self.navigationController.hidesBarsWhenKeyboardAppears=YES;
self.navigationController.hidesBarsOnSwipe=YES;
```

hidesBarsWhenVerticallyCompact 属性设置为 YES，则当设备横屏时导航栏和工具栏会自动隐藏。hidesBarsOnTap 属性设置 YES，则当用户单击屏幕时导航栏和工具栏会自动隐藏。hidesBarsWhenKeyboardAppears 属性设置为 YES，则当键盘弹起时导航栏和工具栏会自动隐藏。hidesBarsOnSwipe 属性设置为 YES，则当用户滑动屏幕时，导航栏和工具栏会自动隐藏。

3.2 标签控制器——UITabBarController

与导航控制器类似，标签控制器也是视图控制器的控制器，是一个容器类。不同的是，标签控制器中管理的视图控制器并不存在层级关系，而是并列的。因此一个视图控制器一旦被加入标签控制器容器中，只要标签控制器存在，视图控制器就不会被释放。标签控制器多用于分类并列结构的应用中。

3.2.1 标签控制器的工作原理

标签控制器中管理的各个视图控制器是并列结构的，如图 3-34 所示。在 UITabBarController 进行初始化的时候，可以为其设置管理的视图控制器。在用户单击标签控制器的标签时会进行视图控制器的切换。

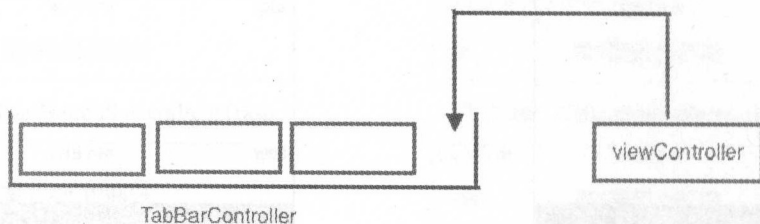


图 3-34 UITabBarController 的工作原理

3.2.2 标签控制器的基础用法解析

使用 Xcode 创建一个名为 UITabBarControllerTest 的工程，可以通过使用纯代码和 storyboard 两种方式来创建标签控制器，在导航控制器一节中我们使用了 storyboard 的创建方法，这里使用纯代码的方式来创建标签控制器。在 ViewController.m 文件中添加如下方法：

```

-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{
    UITabBarController * tabBar = [[UITabBarController alloc] init];
    NSMutableArray * array = [[NSMutableArray alloc] init];
    for (int i=0; i<5; i++) {
        UIViewController * con = [[UIViewController alloc] init];
        con.view.backgroundColor = [UIColor colorWithRed:arc4random()%255/
255.0 green:arc4random()%255/255.0 blue:arc4random()%255/255.0 alpha:1];
        con.tabBarItem.title = [NSString stringWithFormat:@"%d 视图", i];
        con.tabBarItem.image = [UIImage imageNamed:@"tab_bar_icon_home"];
        con.tabBarItem.selectedImage = [UIImage imageNamed:@"tab_bar_icon_
home_selected"];
        [array addObject:con];
    }
    tabBar.viewControllers = array;
    [self presentViewController:tabBar animated:YES completion:nil];
}

```

上面代码设置在单击屏幕时切换出一个标签控制器。for 循环创建了 5 个视图控制器，将这 5 个视图控制器组成的数组赋值给 UITabBarController 的 viewControllers 属性。如果一个 UIViewController 对象被加入进标签控制器管理，则它的 tabBarItem 属性对应于标签控制器上的一个标签。tabBarItem 的 title 属性用于设置标签上的标题，tabBarItem 的 image 属性设置标签的图片，tabBarItem 的 selectedImage 属性设置标签被选中时的图片。运行工程，效果如图 3-35 所示。

UITabBarController 自带一个标签栏，会显示在界面的底部。从图 3-35 中可以看出，当向 UITabBarController 中添加 5 个视图控制器时，标签栏上的标签已经略显拥挤，事实上，如果向标签控制器中添加多于 5 个视图控制器时，UITabBarController 会自动创建一个包含表格视图的导航控制器，在表格视图中将多出的视图控制器排列出来。例如将上面代码的 for 循环条件改为循环 10 次，再次运行工程，效果如图 3-36 所示。

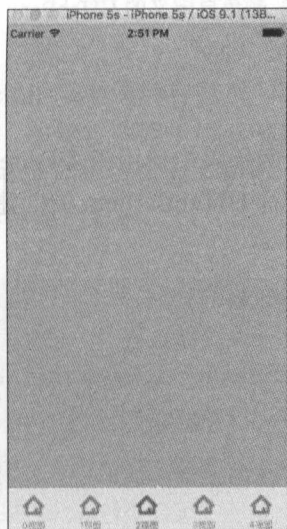


图 3-35 UITabBarController

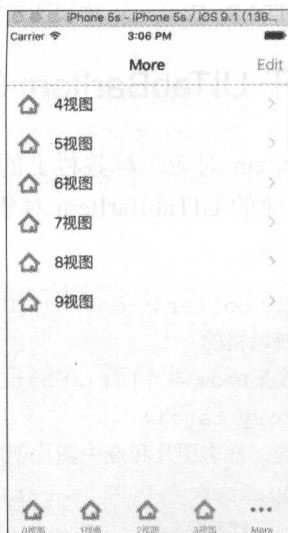


图 3-36 标签数大于 5 的 UITabBarController

从图 3-36 可以看出，第 5 个标签被系统征用作为 more 标签。对应的，系统创建了一个导航控制器，管理从第 5 个视图控制器开始往后的全部视图控制器。除此之外，系统还自动在导航栏上创建了一个 Item 按钮，单击这个按钮时，UITabBarController 会进入编辑状态，用户可以通过拖动的方式改变其中视图控制器的排序，如图 3-37 所示。

在 UI 方面，如下代码可以设置标签栏的背景颜色及上面标签的渲染颜色，标签的渲染颜色会影响到标题与图片。

```
UITabBarController * tabBar = [[UITabBarController alloc] init];
tabBar.tabBar.barTintColor = [UIColor greenColor];
tabBar.tabBar.tintColor = [UIColor purpleColor];
```

运行工程，效果如图 3-38 所示。

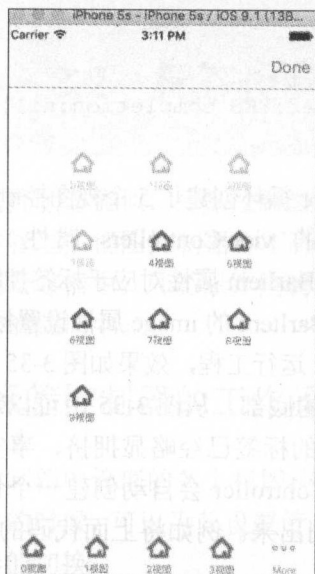


图 3-37 编辑 UITabBarController 中视图控制器的位置

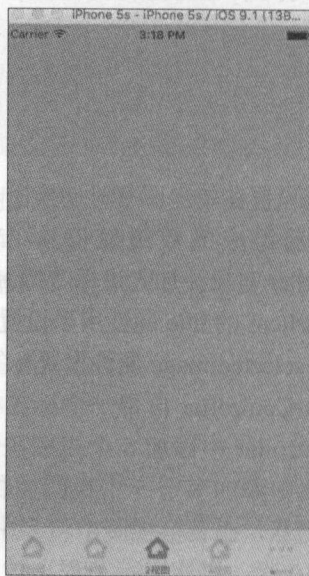


图 3-38 颜色自定义的 UITabBar

3.2.3 关于 UITabBarItem 的使用

UITabBarItem 对应于标签栏上的具体标签，在上面的示例代码中使用了 UINavigationController 中系统自动创建的 UITabBarItem 对象，开发者也可以自己进行 UITabBarItem 的创建，几种创建方式如下所示。

```
UIViewController * con = [[UIViewController alloc] init];
//创建系统风格的
con.tabBarItem = [[UITabBarItem alloc] initWithTabBarSystemItem:UITabBarSystemItemHistory tag:1];
//通过标题、常态图片和选中图片创建
con.tabBarItem = [[UITabBarItem alloc] initWithTitle:@"title" image:[UIImage imageNamed:@"image"] selectedImage:[UIImage imageNamed:@"image"]];
//通过标题和常态图片创建
```



```
con.tabBarItem = [[UITabBarItem alloc] initWithTitle:@"title" image:[UIImage imageNamed:@"image"] tag:1];
```

`initWithTabBarItem:tag:`方法创建一个系统风格的标签,系统提供了许多定义好的风格供开发者自由选用。`initWithTitle:image:selectedImage:`方法第 1 个参数设置标签的标题,第 2 个参数设置常态下的标签图片,第 3 个参数设置选中状态下的标签图片。

使用 `initWithTabBarItem:tag:`方法时,标签的系统风格枚举意义如下:

```
typedef NS_ENUM(NSInteger, UITabBarItem) {
    UITabBarItemMore,           //更多风格
    UITabBarItemFavorites,      //喜爱风格
    UITabBarItemFeatured,      //关注风格
    UITabBarItemTopRated,      //排行风格
    UITabBarItemRecents,       //最近记录风格
    UITabBarItemContacts,      //联系人风格
    UITabBarItemHistory,       //历史风格
    UITabBarItemBookmarks,     //书签风格
    UITabBarItemSearch,        //搜索风格
    UITabBarItemDownloads,     //下载风格
    UITabBarItemMostRecent,    //记录列表风格
    UITabBarItemMostViewed,    //浏览列表风格
};
```



提示

系统风格的 `UITabBarItem` 会提供特定的图片和标题,开发者可以选择适用的直接使用。

3.3 滚动视图——UIScrollView

`UIScrollView` 是所有复杂 UI 视图的基础,表格视图和集合视图都是基于 `UIScrollView` 功能扩展的。滚动视图用于展示某些内容超出一屏的视图,举个最简单的例子,当用户浏览网页时,很少有网站可以仅靠手机小小的一屏将内容展示完全,用户需要在屏幕上进行滑动来使更多的内容替换进屏幕。

3.3.1 使用 UIScrollView 展示视图内容

使用 Xcode 创建一个名为 `UIScrollViewTest` 的工程,在 `ViewController.m` 文件的 `viewDidLoad` 方法中添加如下代码:

```
- (void)viewDidLoad {
    [super viewDidLoad];
```



```

UIScrollView * scrollView = [[UIScrollView alloc] initWithFrame:self.view.frame];
UIImageView * imageView = [[UIImageView alloc] initWithFrame:CGRectMake(40, 50, 240, 400)];
imageView.image = [UIImage imageNamed:@"image"];
[scrollView addSubview:imageView];
scrollView.contentSize=CGSizeMake(self.view.frame.size.width*2, self.view.frame.size.height*2);
[self.view addSubview:scrollView];
}

```

上面代码创建了一个 UIImageView 对象用于显示一张图片，将 UIImageView 添加到了滚动视图 UIScrollView 对象上。有一点需要注意，UIScrollView 的 `contentSize` 属性设置滚动视图的内容大小，内容区域决定可滚动的范围。这时运行工程，使用手指在屏幕上进行滑动操作，会看到图片随着手指的移动而进行移动。

在滚动视图进行滚动时，可以看到屏幕的右侧和下侧有滚动进度相关的指示条，如果将滚动视图拖动到某一极限后还继续拖动的话，此时松手会出现回弹效果，滚动视图的这些细节都可以通过相应接口进行自定义，代码示例如下：

```

scrollView.bounces=YES;
scrollView.alwaysBounceHorizontal=YES;
scrollView.alwaysBounceVertical =YES;
scrollView.pagingEnabled=YES;
scrollView.showsHorizontalScrollIndicator=YES;
scrollView.showsVerticalScrollIndicator=YES;
scrollView.indicatorStyle=UIScrollViewIndicatorStyleBlack;
scrollView.scrollsToTop=YES;

```

`bounces` 属性用于设置是否开启回弹效果，包括水平方向的回弹效果和竖直方向的回弹效果。`alwaysBounceHorizontal` 属性设置是否始终开启水平方向的回弹效果。`alwaysBounceVertical` 属性设置是否始终开启竖直方向的回弹效果。`pagingEnabled` 属性设置是否开启分页效果，开启分页效果后，对 UIScrollView 进行滑动操作时会有自动定位的功能。`showsHorizontalScrollIndicator` 设置是否显示水平提示条，`showsVerticalScrollIndicator` 设置是否显示竖直提示条。`indicatorStyle` 属性设置提示条的风格，支持的枚举有如下 3 种：

```

typedef NS_ENUM(NSInteger, UIScrollViewIndicatorStyle) {
    UIScrollViewIndicatorStyleDefault,    // 默认风格
    UIScrollViewIndicatorStyleBlack,      // 黑色风格
    UIScrollViewIndicatorStyleWhite       // 白色风格
};

```

UIScrollView 的 `scrollsToTop` 属性也十分有用，如果设置为 YES，则当用户单击屏幕上方的状态栏时，滚动视图会自动滚动到顶端，设置为 NO 则无此特性。



提示

状态栏是指屏幕上显示服务商和时间的那一栏。

3.3.2 UIScrollView 的代理方法

在用户对滚动视图进行操作时，UIScrollViewDelegate 协议中定义了许多方法可以帮助开发者对用户的行为进行监听。首先要让相应类遵守 UIScrollViewDelegate 协议，设置 UIScrollView 对象的 delegate 属性为遵守协议的类对象，协议中常用的方法与其意义如下：

```

-(BOOL)scrollViewShouldScrollToTop:(UIScrollView *)scrollView{
    return YES;
}
//滚动视图已经滚动到顶端时触发的方法
-(void)scrollViewDidScrollToTop:(UIScrollView *)scrollView{

}
//将要开始拖动滚动视图时触发的方法
-(void)scrollViewWillBeginDragging:(UIScrollView *)scrollView{

}
//滚动视图将要结束拖动时触发的方法
-(void)scrollViewWillEndDragging:(UIScrollView *)scrollView withVelocity:
(CGPoint)velocity targetContentOffset:(inout CGPoint *)targetContentOffset{

}
//滚动视图将要减速时触发的方法
-(void)scrollViewWillBeginDecelerating:(UIScrollView *)scrollView{

}
//滚动视图将要缩放时触发的方法
-(void)scrollViewWillBeginZooming:(UIScrollView *)scrollView withView:(UI
View *)view{

}
//滚动视图已经缩放时触发的方法
-(void)scrollViewDidZoom:(UIScrollView *)scrollView{

}
//滚动视图已经结束拖拽时触发的方法
-(void)scrollViewDidEndDragging:(UIScrollView *)scrollView willDecelerate:
(BOOL)decelerate{

}

```

```

//滚动视图已经结束减速时触发的方法
-(void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView{

}
//滚动视图滚动动画结束时调用
-(void)scrollViewDidEndScrollingAnimation:(UIScrollView *)scrollView{

}
//滚动视图开始滚动时调用
-(void)scrollViewDidScroll:(UIScrollView *)scrollView{

}
//设置进行缩放的子视图
-(UIView *)viewForZoomingInScrollView:(UIScrollView *)scrollView{
    return scrollView.subviews.firstObject;
}

```

上面的方法中，`scrollViewShouldToTop`方法在用户单击状态栏后会被调用，返回值决定是否进行滑动到顶端的操作，返回 YES 则是允许，返回 NO 则是不允许。`ScrollViewDidScrollToTop`方法在滚动视图滚动到顶端后被调用。`scrollViewWillBeginDragging`方法在手指将要拖动屏幕时被调用。这些方法中 `scrollViewDidScroll`方法应用得最多，开发者通常会在该方法中监听滚动视图当前的位置。`viewForZoomingInScrollView`方法十分有趣，这里将返回一个滚动视图的子视图作为返回值，当用户在滚动视图上进行两指捏合操作时，相应的视图会被放大或者缩小，这个功能十分类似于系统自带的相册，在浏览相片的时候相片可以进行放大缩小，与这个方法相配合，`UIScrollView` 中关于缩放操作有如下属性可以定义：

```

scrollView.minimumZoomScale =0.5;
scrollView.maximumZoomScale =2.0;
scrollView.bouncesZoom =YES;

```

`minimumZoomScale` 属性设置最小缩小倍率，`maximumZoomScale` 设置最大放大倍率。`bouncesZoom` 属性设置缩放是否支持回弹效果。



提示

在模拟器中如果需要使用双指捏合的操作，需要按住键盘 option 键。

3.4 网络视图——UIWebView

在一款应用软件中，经常会嵌套一些网页，甚至有些应用程序除了主框架，其他都是由网页视图构成的。这种做法是有很大优势的，比如你需要在应用中添加一个新的功能，如果是完全通过原生框架开发的软件，新加功能需要经过开发、提交、审核上线的过程才能被用户使用，这对时效性要求很高的需求来说是无法接受的。但是如果你需要加的功能其模块是通过网络视

图实现的，则只需要改 Web 页的内容，随时就可以供用户使用。在另一方面，网络视图和原生的比也有其无法避免的缺陷，例如必须在用户网络良好的环境下使用，用户交互不如原生流畅与友好等。

3.4.1 App 网络传输安全策略

随着 iOS 9 的推出和 Xcode 升级到 7.0 版本，在开发有关网络数据传输的应用程序时，将默认采用 Https 类型的连接。目前很多网站依然采用的是 Https 类型的连接，在 Xcode 中如果不进行一些兼容性的设置，这类网站将无法加载。

使用 Xcode 创建一个名为 UIWebViewTest 的工程，单击工程 Info.plist 配置文件，如图 3-39 所示。

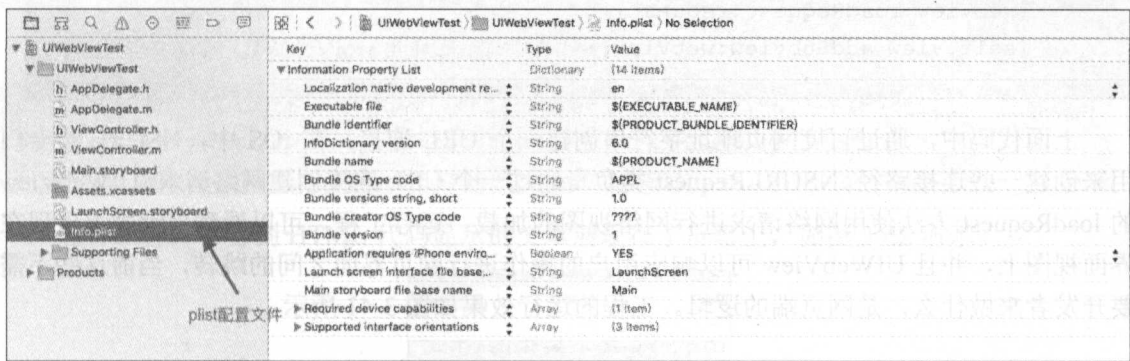


图 3-39 工程配置文件

在 information Property List 中添加一个新的键，选择 App Transport Security Settings，这是一个字典类型的键值集合，为其中添加一个名为 Allow Arbitrary Loads 的 Boolean 类型的键，将其值设置为 YES。如图 3-40 所示。

Key	Type	Value
Information Property List	Dictionary	(15 items)
App Transport Security Settings	Dictionary	(1 item)
Allow Arbitrary Loads	Boolean	YES
Localization native development re...	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(3 items)

图 3-40 配置兼容 http 类型传输的键值对

通过上面的配置，工程中就可以使用以 Https 类型连接的网络请求了。

3.4.2 通过网络请求加载 UIWebView

通过网络请求加载 UIWebView 即是在 UIWebView 上显示网页内容，类似于手机上的网页浏览器。使用 UIWebViewTest 工程在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIWebView * webView = [[UIWebView alloc] initWithFrame:self.view.frame];
    NSURL * url = [NSURL URLWithString:@"http://www.baidu.com"];
    NSURLRequest * request = [NSURLRequest requestWithURL:url];
    [webView loadRequest:request];
    [self.view addSubview:webView];
}
```

上面代码中，通过百度网页地址字符串创建一个 URL 连接，在 iOS 中，NSURL 类专门用来创建一些连接路径。NSURLRequest 类负责通过一个 URL 连接创建网络请求，UIWebView 的 loadRequest: 方法使用网络请求进行网络视图的加载。运行工程，可以看到百度网页展现在界面视图上，并且 UIWebView 可以响应用户的操作进行网页界面之间的跳转，当前这些不需要开发者来做什么，是网页端的逻辑。工程的运行效果如图 3-41 所示。



图 3-41 使用 UIWebView 加载百度首页

3.4.3 通过 HTML 字符串加载 UIWebView

大部分的网页都是通过 HTML 语言来开发的，UIWebView 类中也提供了一个方法通过 HTML 字符串进行数据加载操作。有了这个方法，使加载本地的网页文件成为可能。在实际

应用中,无论是开发缓存使应用可以离线进行网页视图的展示还是远程下载 HTML 文件使应用程序可以进行热点更新,使用这种方式加载 UIWebView 都扮演了十分重要的角色。

下面这段 HTML 代码用于简单地显示一行 hello world 文本。

```
<html>
<head>
<meta charset="UTF-8">
<title> 主标题 | 副标题</title>
</head>
<body>
<p>hello world</p>
</body>
</html>
```

将工程中加载 UIWebView 的代码改写如下:

```
NSString *htmlStr = @"<html><head><meta charset=\"UTF-8\"><title> 主标题 |
副标题</title></head><body><p>hello world</p></body></html>";
[webView loadHTMLString:htmlStr baseURL:nil];
```

运行工程,将看到 HTML 代码显示的文本展现在了屏幕上,如图 3-42 所示。

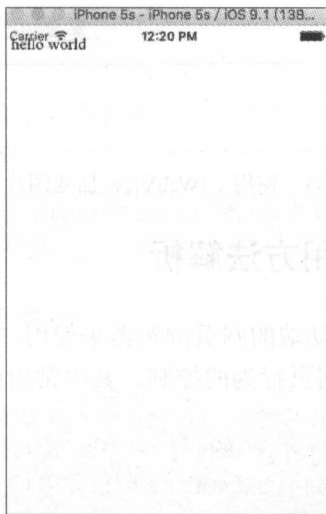


图 3-42 使用 HTML 字符串加载 UIWebView



提示

对于 HTML 字符串中的引号,要使用转义符进行转义。

3.4.4 通过 NSData 数据加载 UIWebView

通过 NSData 进行 UIWebView 的加载是加载 UIWebView 的 3 种方法中最灵活自由的。使用这种方式时不再限定数据的格式,因此,许多 gif 动态图片的展示也都采用这种方式。使用如下的代码可以使用 UIWebView 来加载图片数据:

```

    NSURL * imageUrl = [[NSURL alloc] initWithFileURLWithPath:[NSBundle mainBundle]
pathForResource:@"image" ofType:@"png"]];
    NSData * data = [NSData dataWithContentsOfURL:imageUrl];
    [webView loadData:data MIMEType:@"image/gif" textEncodingName:nil baseURL:nil];

```

运行效果如图 3-43 所示。

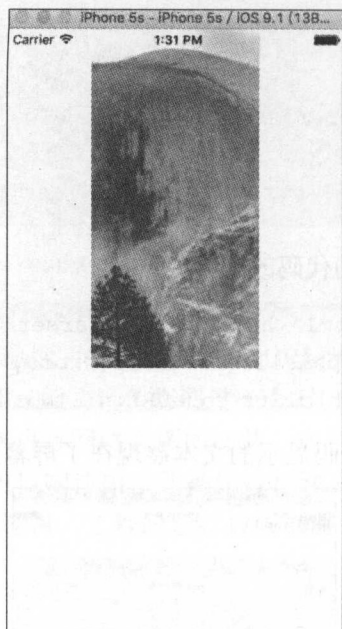


图 3-43 使用 UIWebView 加载图片数据

3.4.5 UIWebView 中常用方法解析

UIWebView 可以充当一个移动端的网页浏览器来使用，UIWebView 类中也提供了一些方法，可以很好地帮助开发者进行网页行为的控制。其中常用的属性和方法如下所示：

```

- (void)reload;
- (void)stopLoading;
- (void)goBack;
- (void)goForward;
@property (nonatomic, readonly, getter=canGoBack) BOOL canGoBack;
@property (nonatomic, readonly, getter=canGoForward) BOOL canGoForward;
@property (nonatomic, readonly, getter=isLoading) BOOL loading;
@property (nonatomic) BOOL scalesPageToFit;
@property (nonatomic) UIDataDetectorTypes dataDetectorTypes;

```

调用 reload 方法将重新加载 UIWebView 对象，用于刷新当前界面。stopLoading 方法用于停止当前页面的加载。属性 loading 会返回一个 BOOL 值，告诉开发者当前 UIWebView 对象的状态是否正在加载数据。goBack 方法用于退回到上一界面，类似浏览器中的返回操作，当

然这个操作的前提是有上一个界面,如果本来就在初始的页面,是不能退回的,属性 `canGoBack` 用于判断是否可以进行退回操作。与 `goBack` 方法对应, `goForward` 用于前进到下一界面,同样必须在回退过之后才能调用这个方法进行前进, `canGoForward` 属性用于判断是否可以前进操作。 `scalesPageToFit` 属性用于设置界面的尺寸是否适配 `UIWebView` 对象的尺寸。 `dataDetectorTypes` 属性用于设置一些可以进行用户交互的字段类型,例如当用户使用 `UIWebView` 进行网页浏览时,网页上出现一个电话号码,这时用户可能需要拨打这个电话, `UIWebView` 支持探测出电话号码的字段,当用户单击这个电话号码时会自动跳转到手机拨号的界面。 `UIDataDetectorTypes` 枚举支持的探测类型如下所示:

```
typedef NS_OPTIONS(NSUInteger, UIDataDetectorTypes) {
    UIDataDetectorTypePhoneNumber = 1 << 0,           // 探测电话号码
    UIDataDetectorTypeLink = 1 << 1,                 // 探测网页链接
    UIDataDetectorTypeAddress = 1 << 2,               // 探测地址
    UIDataDetectorTypeCalendarEvent = 1 << 3,         // 探测日历事件
    UIDataDetectorTypeNone = 0,                       // 不进行探测
    UIDataDetectorTypeAll = NSUIntegerMax              // 探测所有类型
};
```



提示

`loading`、`canGoBack`、`canGoForward` 这 3 个属性是只读的,只能进行 `get` 方法获取,不能 `set` 方法设置。

对于以 `NS_OPTIONS` 标记的枚举是支持进行或运算的,例如 `UIWebView` 对象设置探测电话号码和网页链接,使用如下代码:

```
webView.dataDetectorTypes = UIDataDetectorTypePhoneNumber|UIDataDetectorTypeLink;
```

3.4.6 UIWebView 的代理方法

在 `UIWebViewDelegate` 中定义了一些回调方法,这些回调方法可以返回当前网络视图的状态。为 `ViewController` 类添加遵守 `UIWebViewDelegate` 协议的代码并将 `UIWebView` 对象的 `delegate` 属性设置为 `ViewController` 对象。 `UIWebViewDelegate` 中方法如下:

```
//加载失败调用的方法
-(void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error{
}

//将要加载请求时调用的方法
-(BOOL)webView:(UIWebView *)webView shouldStartLoadWithRequest:(NSURLRequest *)request navigationType:(UIWebViewNavigationType)navigationType{
    return YES;
}

//加载数据结束调用的方法
-(void)webViewDidFinishLoad:(UIWebView *)webView{
```



```

}
// 已经开始加载数据时调用的方法
- (void)webViewDidStartLoad:(UIWebView *)webView{
}

```

其中 `webView:shouldStartLoadWithRequest:navigationType:` 方法有一个 `BOOL` 类型的返回值，当返回 `YES` 时表示允许加载请求，返回 `NO` 表示不允许加载请求。

3.5 表格视图——UITableView

无论做哪个平台的软件开发，表格视图都是十分常用的一个 UI 视图。在 iOS 中，表格视图由 `UITableView` 这个类来提供支持。`UITableView` 内部封装了一套复用机制，通过复用，开发者可以高效地展示数据量非常大的列表而不用担心内存方面的问题。

3.5.1 UITableView 的创建与复用机制

与先前介绍的 UI 控件有很大的不同，`UITableView` 通过数据源来进行视图与数据的绑定，通过代理方法来对 `UITableView` 中的一些属性进行设置。在 `UITableView` 中每条数据通过 `UITableViewCell` 类对象来展示，`UITableView` 控件对其中的 cell 进行复用管理。`UITableView` 的创建关系如图 3-44 所示。

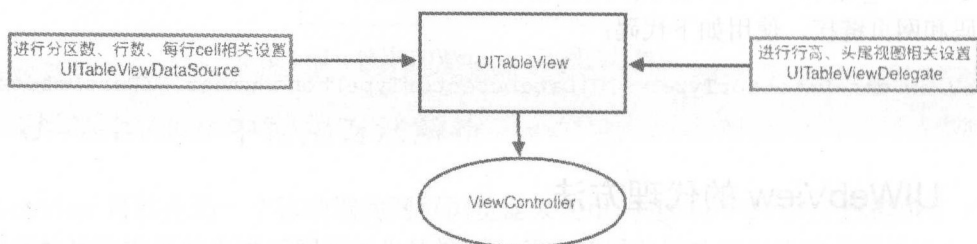


图 3-44 UITableView 的创建关系

在使用 `UITableView` 控件时，`UITableViewDataSource` 协议中的两个方法必须实现，其他方法可选实现，`UITableViewDelegate` 协议中的所有方法都可选实现。`UITableViewDataSource` 协议中必须实现的两个方法如下：

```

// 设置表格视图有多少行
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section{
    return 1;
}
// 设置每行的 UITableViewCell
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath{

```

```

UITableViewCell * cell = [[UITableViewCell alloc] initWithStyle:UITableView
UITableViewCellStyleDefault reuseIdentifier:@"cellID"];
return cell;
}

```

tableView:numberOfRowsInSection: 方法设置表格视图中的行数，这个方法中会传入一个分区的参数，可以通过这个参数判断出不同的分区设置不同的行数。**tableView:cellForRowAtIndexPath:** 方法用于设置展示每一行数据的 cell 视图，这个方法中的 indexPath 参数中包含分区与行数的信息。

UITableView 对其中 cell 的复用是采取复用池的设计模式。例如，一个表格视图有 100 行数据，视图上每屏可以显示 10 行数据，那么 UITableView 实际上创建其上 cell 视图的时候只需要创建 11 个 cell 视图即可够用，当 cell 被滑出屏幕外则它被回收进复用池，新的 cell 将要滑入进屏幕时从复用池中取用。UITableView 对 cell 的复用机制如图 3-45 所示。

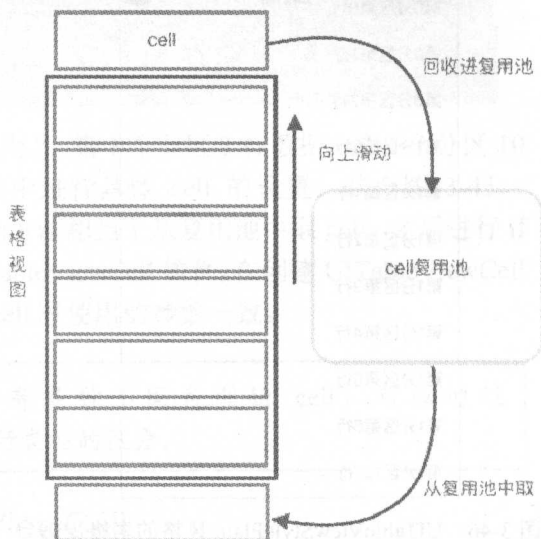


图 3-45 UITableView 中 cell 的复用机制

3.5.2 创建一个表格视图 UITableView

使用 Xcode 创建一个名为 UITableViewTest 的工程，在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码：

```

- (void)viewDidLoad {
    [super viewDidLoad];
    UITableView * tableView = [[UITableView alloc] initWithFrame:self.view.
frame style:UITableViewStylePlain];
    tableView.delegate=self;
    tableView.dataSource=self;
    [self.view addSubview:tableView];
}

```

上面代码中使用 initWithFrame:style: 方法来初始化 UITableView 控件，这里的风格参数枚举值有两种。

```

typedef NS_ENUM(NSInteger, UITableViewStyle) {
    UITableViewStylePlain,           // 规范的表格视图风格
    UITableViewStyleGrouped         // 分组的表格视图风格
};

```

UITableViewStylePlain 创建标准风格的表格视图，UITableViewStyleGrouped 创建分组风格的表格视图，其区别在于如果有多个分区，UITableViewStyleGrouped 风格创建的表格视图每个分区间将有一定间距。如图 3-46 与图 3-47 所示。

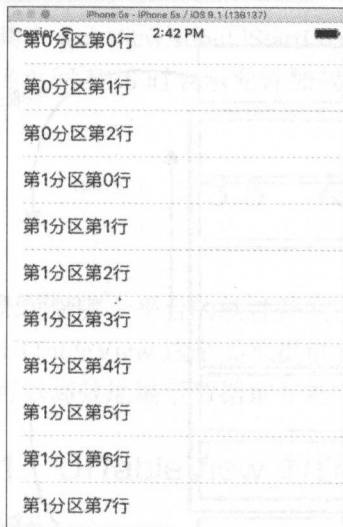


图 3-46 UITableViewStylePlain 风格的表格视图

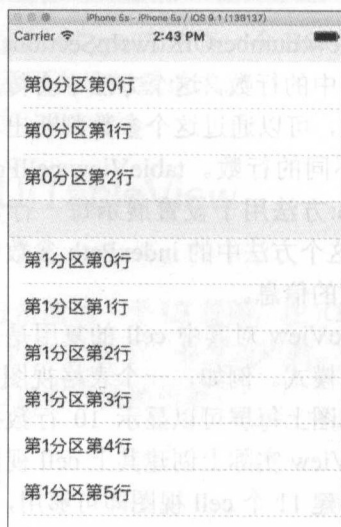


图 3-47 UITableViewStyleGrouped 风格的表格视图

在 ViewController.m 文件中添加遵守相关协议的代码：

```
@interface ViewController ()<UITableViewDataSource,UITableViewDelegate>
@end
```

在 ViewController.m 文件中实现相关的协议方法：

```
//设置表格视图有多少行
-(NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section{
    if (section==0) {
        return 3;
    }else{
        return 10;
    }
}

-(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView{
    return 2;
}

//设置每行的 UITableViewCell
-(UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath{
    UITableViewCell * cell = [tableView dequeueReusableCellWithIdentifier:@"cellID"];
```



```

    if (cell==nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"cellID"];
    }
    cell.textLabel.text = [NSString stringWithFormat:@"第%d分区第%d行", indexPath.section, indexPath.row];
    return cell;
}

```

上面的协议方法中，将表格视图的分区数设置为 2，第 1 个分区 3 行数据，第 2 个分区 10 行数据。在 `tableView:cellForRowAtIndexPath:` 方法中进行具体 `cell` 的设置，结合图 3-45，`UITableView` 类的 `dequeueReusableCellWithIdentifier:` 方法相当于从复用池中取 `cell`，之后进行 `if` 判断，如果复用池中 `cell` 不够，则创建一个新的 `UITableViewCell` 控件，在创建 `UITableViewCell` 控件时设置的 `Identifier` 参数必须和从复用池中取 `cell` 时使用的参数一致。



提示

一个 `UITableView` 控件中可以有多种不同类型的 `cell`，可以通过 `UITableViewCell` 的 `Identifier` 来进行类型的区分。

3.5.3 关于表格数据的载体 UITableViewCell

关于 `UITableViewCell`，系统提供了几种风格样板。在使用 `initWithStyle:reuseIdentifier:` 方法进行创建的时候，可选的风格枚举如下所示：

```

typedef NS_ENUM(NSInteger, UITableViewCellStyle) {
    UITableViewCellStyleDefault,    // 默认风格，内置一个 UILabel 和 UIImageView
    UITableViewCellStyleValue1,     // 设置应用中 cell 的风格
    UITableViewCellStyleValue2,     // 联系人应用中 cell 的风格
    UITableViewCellStyleSubtitle    // 有一个子标题的 cell 风格
};

```

更多情况下，开发者会采用继承自 `UITableViewCell` 的自定义 `cell` 来进行表格数据的展示。`UITableViewCell` 中还有许多属性可以用来对其 UI 进行一些自定义的设置，示例代码如下：

```

-(UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
(NSIndexPath *)indexPath{
    UITableViewCell * cell = [tableView dequeueReusableCellWithIdentifier:
@"cellID"];
    if (cell==nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle reuseIdentifier:@"cellID"];
    }
    cell.textLabel.text = [NSString stringWithFormat:@"第%d分区", indexPath.section];
}

```



```

cell.detailTextLabel.text = [NSString stringWithFormat:@"第%d行", index
Path.row];
cell.imageView.image= [UIImage imageNamed:@"image"];
cell.backgroundColor = [UIColor purpleColor];
cell.accessoryType = UITableViewCellAccessoryCheckmark;
return cell;
}

```

UITableViewCell 的 `titleLabel` 属性是 cell 控件的主标题标签，是 UILabel 类型的，可以使用 UILabel 类中的方法对其进行设置。`detailTextLabel` 属性在 UITableViewCell 的风格为 UITableViewCellStyleSubtitle 时有效，为 cell 控件的副标题标签。`imageView` 属性是 cell 控件上的图片视图。`accessoryType` 属性设置 cell 控件的附加视图类型，附加视图会出现在 cell 的最右侧，一般会是一个功能按钮的模样，UITableViewCellAccessoryType 有如下几种枚举值：

```

typedef NS_ENUM(NSInteger, UITableViewCellAccessoryType) {
    UITableViewCellAccessoryNone,                //无附加视图
    UITableViewCellAccessoryDisclosureIndicator, //规则的前进箭头样式
    UITableViewCellAccessoryDetailDisclosureButton, //复合样式
    UITableViewCellAccessoryCheckmark,           //对号图标样式
    UITableViewCellAccessoryDetailButton         // 详情按钮样式
};

```

各种风格的效果如图 3-48~图 3-51 所示。

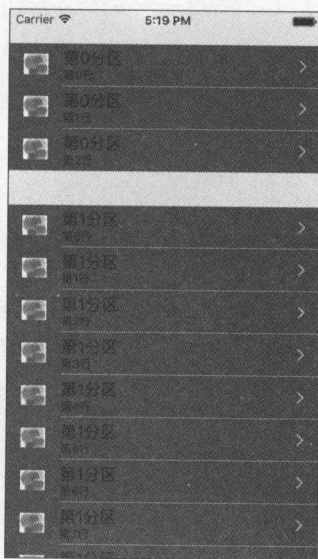


图 3-48 UITableViewCellAccessoryDisclosureIndicator

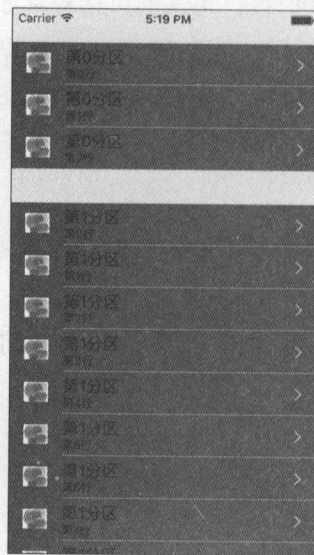


图 3-49 UITableViewCellAccessoryDetailDisclosureButton

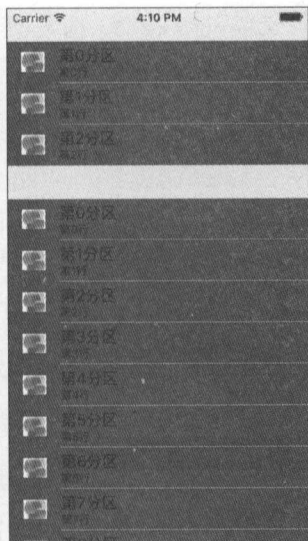


图 3-50 UITableViewCellAccessoryCheckmark

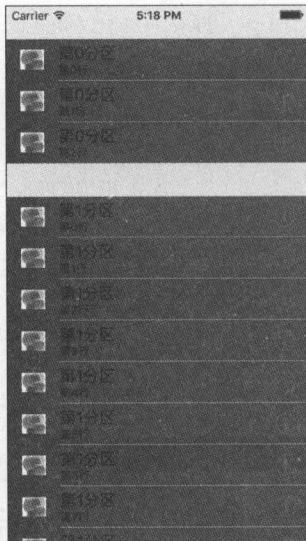


图 3-51 UITableViewCellAccessoryDetailButton

3.5.4 设置 UITableView 的行高和头尾视图

UITableView 的具体行高度和头尾视图的设置, 是通过 UITableViewDelegate 协议中约定的方法来完成的, 在上面工程的 ViewController.m 中继续添加如下这些方法。

```
//设置行高
-(CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath{
    if (indexPath.section==0) {
        return 100;
    }else{
        return 44;
    }
}
//设置分区尾视图高度
-(CGFloat)tableView:(UITableView *)tableView heightForFooterInSection:(NSInteger)section{
    return 50;
}
//设置分区头视图高度
-(CGFloat)tableView:(UITableView *)tableView heightForHeaderInSection:(NSInteger)section{
    return 50;
}
//设置分区的尾视图
-(UIView *)tableView:(UITableView *)tableView viewForFooterInSection:(NSInteger)section{
```

```

    UIView * view = [[UIView alloc] initWithFrame:CGRectMake(0, 0, self.view.frame.size.width, 50)];
    view.backgroundColor = [UIColor greenColor];
    return view;
}
//设置分区的头视图
-(UIView *)tableView:(UITableView *)tableView viewForHeaderInSection:(NSInteger)section{
    UIView * view = [[UIView alloc] initWithFrame:CGRectMake(0, 0, self.view.frame.size.width, 50)];
    view.backgroundColor = [UIColor blueColor];
    return view;
}

```

上面代码中，将第 1 个分区中的行高设置为 100 个单位，将第 2 个分区中的行高设置成了 44 个单位。设置每个分区的头尾视图高度都是 50 个单位，并且将头视图设置成了蓝色背景的视图，将尾视图设置成了绿色背景的视图。运行工程，效果如图 3-52 所示。

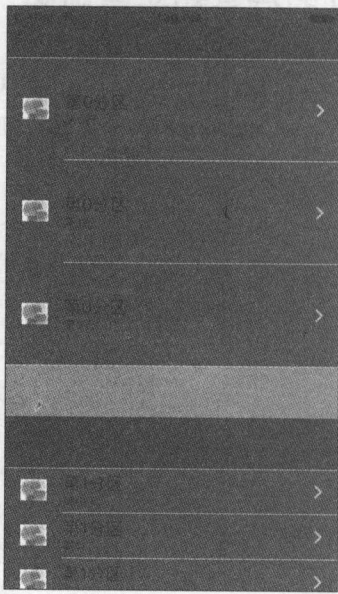


图 3-52 设置 UITableView 的头尾视图和行高

3.5.5 UITableView 的用户交互行为

UITableView 的核心功能是与用户进行交互，例如资讯类应用中表格视图可能用于展示每篇咨询的标题，单击后界面会跳转到咨询的详情；单纯的列表也可能会需要有增、删、移动数据顺序等操作。对于这些用户交互的操作，UITableView 都提供了方便易用的接口或者协议方法。

当用户选中一条 cell 时，会调用如下的代理方法：


```
//选中 cell 时调用的方法
-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath{

}
```

tableView:didSelectRowAtIndexPath:方法会将用户选中 cell 所在的分区和行位信息传递进来。

对于 UITableView 中 cell 的增、删和移动操作,需要将 UITableView 控件设置为编辑模式,使用如下代码设置 UITableView 对象:

```
tableView.editing=YES;
```

需要实现如下的代理方法来设置表格视图中具体 cell 的编辑模式类型:

```
/设置 cell 的编辑模式类型
-(UITableViewCellEditingStyle)tableView:(UITableView *)tableView editingStyleForRowAtIndexPath:(NSIndexPath *)indexPath{
    if (indexPath.section==0) {
        return UITableViewCellEditingStyleInsert;
    }else{
        return UITableViewCellEditingStyleDelete;
    }
}
```

这个代理方法需要返回一个 UITableViewCellEditingStyle 类型的枚举,枚举值意义如下:

```
typedef NS_ENUM(NSUInteger, UITableViewCellEditingStyle) {
    UITableViewCellEditingStyleNone,           //没有编辑类型
    UITableViewCellEditingStyleDelete,        //删除风格的编辑类型
    UITableViewCellEditingStyleInsert         //插入风格的编辑类型
};
```

如果需要 UITableView 中的 cell 支持位置移动操作,需要实现如下两个代理方法:

```
//设置 cell 是否支持位置移动
-(BOOL)tableView:(UITableView *)tableView canMoveRowAtIndexPath:(NSIndexPath *)indexPath{
    return YES;
}

//位置移动后回调的方法
-(void)tableView:(UITableView *)tableView moveRowAtIndexPath:(NSIndexPath *)sourceIndexPath toIndexPath:(NSIndexPath *)destinationIndexPath{

}
```

运行工程,可以看到编辑状态下的 UITableView 效果如图 3-53 所示。

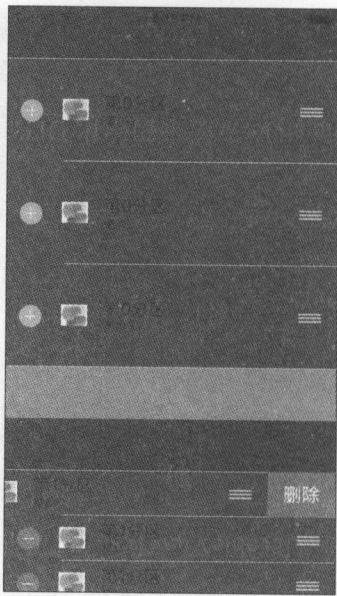


图 3-53 编辑状态下的 UITableView

从图 3-53 可以看到, `UITableViewCellEditingStyleInsert` 风格的 cell 编辑时左侧会显示一个加号按钮, `UITableViewCellEditingStyleDelete` 风格的 cell 编辑时左侧会显示一个减号按钮, 单击减号按钮之后, cell 右侧会滑出一个删除按钮, 删除按钮的标题可以通过下面的代理方法进行设置。

```
//设置删除按钮的标题
-(NSString *)tableView:(UITableView *)tableView titleForDeleteConfirmationButtonForRowAtIndexPath:(NSIndexPath *)indexPath{
    return @"删除";
}
```

当用户单击删除按钮或者插入按钮后, 会调用下面的方法, 开发者可以在下面代理方法进行逻辑处理。

```
/提交编辑操作时触发的方法
-(void)tableView:(UITableView *)tableView commitEditingStyle:(UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)indexPath{
}

```

通过上面方法中传入的 `editingStyle` 参数可以确定是插入还是删除操作, `indexPath` 参数可以确定是具体对哪一个 cell 进行的操作。



提示

UITableView 编辑模式的相关方法都是 UI 层面的方法, 并不会改变数据的结构, 因此开发者需要手动对数据进行操作, 例如删除、添加和改变其顺序等。具体应用会在实战演练中介绍。

3.5.6 为 UITableView 添加索引栏

打开 iPhone 手机的通讯录, 会发现其实通讯录软件就是由一个 UITableView 来实现的, 在这个表格视图中将联系人按姓名拼音排序进行了分区, 每个分区的头视图上标记这个分区的姓名首字母, 并且在视图的右侧还有一个索引栏, 索引栏上有所有字母的索引, 单击相应的索引 UITableView 会直接跳转到相应的分区, 便于用户查找联系人。

实现如下代理方法可以为 UITableView 添加一个分区索引栏:

```
-(NSArray<NSString *> *)sectionIndexTitlesForTableView:(UITableView *)tableView{
    return @[@"one",@"two"];
}
```

sectionIndexTitlesForTableView:方法需要返回一个数组, 数组中必须全部为 NSString 类型的字符串, 这里传入的数组中元素的个数应与 UITableView 控件的分区数一致, 并且他们之间有一一对应的关系。比如, 当单击“two”索引时, UITableView 控件会自动滚动到第 2 个分区处。运行工程, 效果如图 3-54 所示。

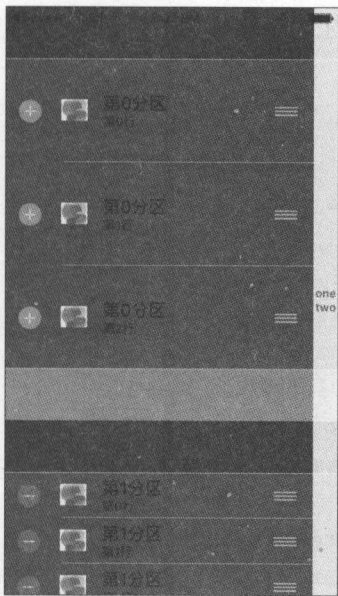


图 3-54 带索引栏的 UITableView

3.6 复杂布局视图——UICollectionView

使用 UITableView 来进行表格视图的搭建是完全没有问题的, 但是 UITableView 也有很多局限性, 对于一些更加复杂的布局样式, 就显得有些力不从心了。例如, UITableView 只允许纵向排列的表格, 不允许横向排列; UITableView 每一行只能有一个数据载体 cell, 不支持在一行中排列多个数据载体。对于这些复杂的布局需求, UICollectionView 可以提供更好的支持。

3.6.1 UICollectionView 控件的优势与布局方式

相比于 UITableView, UICollectionView 有着很大的灵活性和扩展性。其主要优势有如下几点。

- (1) 支持水平方向和竖直方向两个方向的布局。
- (2) 通过 UICollectionViewLayout 类配置的方式进行界面布局。
- (3) 类似与 UITableView 的 cell, UICollectionView 中的数据载体 item 的大小和位置更加灵活。
- (4) 通过 UICollectionViewLayoutDelegate 协议方法可以动态地对布局进行重设。
- (5) 支持完全自定义的 UICollectionViewLayout 子类来进行各种复杂布局。

UICollectionView 通过独立的 Layout 类来进行界面的布局, Layout 类中实际存放的是每个数据载体 item 的布局信息, 包括大小、位置、3D 变换等。

3.6.2 使用 UICollectionView 进行九宫格式的布局

九宫格是一种十分常用的布局模式, 经常会见于功能列表的设计。使用 Xcode 创建一个名为 UICollectionViewTestOne 的工程, 在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UICollectionViewFlowLayout * layout = [[UICollectionViewFlowLayout all
    oc]init];
    layout.scrollDirection=UICollectionViewScrollDirectionVertical;
    layout.itemSize = CGSizeMake(100, 100);
    UICollectionView * collectionView = [[UICollectionView alloc] initWithFram
    e:self.view.frame collectionViewLayout:layout];
    collectionView.backgroundColor = [UIColor whiteColor];
    [collectionView registerClass:[UICollectionViewCell class] forCellWith
    hReuseIdentifier:@"cellID"];
    collectionView.delegate=self;
    collectionView.dataSource=self;
    [self.view addSubview:collectionView];
}
```

上面代码先创建了一个 UICollectionViewFlowLayout 类对 UICollectionView 的布局进行支持, UICollectionViewFlowLayout 是系统提供的一个流式布局类, UICollectionViewFlowLayout 类的 scrollDirection 属性设置布局的方向, 支持的布局方向枚举如下所示:

```
typedef NS_ENUM(NSInteger, UICollectionViewScrollDirection) {
    UICollectionViewScrollDirectionVertical,    // 竖直方向
    UICollectionViewScrollDirectionHorizontal  // 水平方向
};
```


itemSize 属性负责设置 UICollectionView 中每个数据载体 item 的尺寸大小。使用 UICollectionView 类的 initWithFrame:collectionViewLayout:方法对 UICollectionView 对象进行创建,这个方法中第 2 个参数即为对应布局类对象。需要注意的是,UICollectionView 的使用必须注册一个类作为数据载体类,使用 registerClass:forCellWithReuseIdentifier:方法来注册, UICollectionView 内部也会有一套复用机制来对注册的数据载体进行复用。和 UITableView 类似, UICollectionView 也是通过 dataSource 和 delegate 来进行数据填充和相关属性设置。

在 ViewController.m 中添加遵守相关协议的代码如下:

```
@interface ViewController ()<UICollectionViewDataSource, UICollectionViewDelegate>
@end
```

在 ViewController.m 中实现如下的协议方法:

```
-(NSInteger)numberOfSectionsInCollectionView:(UICollectionView *)collectionView{
    return 1;
}

-(NSInteger)collectionView:(UICollectionView *)collectionView numberOfItemsInSection:(NSInteger)section{
    return 10;
}

-(UICollectionViewCell *)collectionView:(UICollectionView *)collectionView cellForItemAtIndexPath:(NSIndexPath *)indexPath{
    UICollectionViewCell * cell = [collectionView dequeueReusableCellWithReuseIdentifier:@"cellID" forIndexPath:indexPath];
    cell.backgroundColor =[UIColor colorWithRed:arc4random()%255/255.0 green:arc4random()%255/255.0 blue:arc4random()%255/255.0 alpha:1];
    return cell;
}
```

UICollectionView 实现的协议方法与使用 UITableView 时实现的协议方法极为类似, numberOfSectionsInCollectionView:方法设置分区数, collectionView:numberOfItemsInSection:设置每个分区数据载体 item 数, collectionView:cellForItemAtIndexPath:方法用于设置具体每条数据载体 item。

运行工程,效果如图 3-55 所示。



提示

流布局又称瀑布流布局,是一种比较流行的网页布局模式,视觉效果多表现为参差不齐的多栏布局。

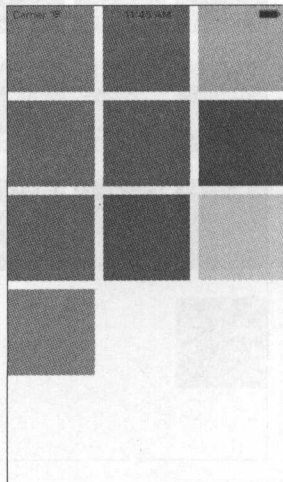


图 3-55 UICollectionView 实现的九宫格布局

3.6.3 创建更加灵活的流式布局

从图 3-55 可以看出，UICollectionView 中每列和每行 item 都会有一定的间距，开发者可以通过设置相关属性来控制间距的最小值，UICollectionView 会自动根据间距最小值和每个 item 的尺寸计算出每行可以排列多少个 item。下面的代码可以设置行列间距：

```
UICollectionViewFlowLayout * layout = [[UICollectionViewFlowLayout alloc]
initWith];
layout.minimumLineSpacing=30;
layout.minimumInteritemSpacing=10;
```

UICollectionViewFlowLayout 的 minimumLineSpacing 属性设置最小的行间距，minimumInteritemSpacing 属性设置最小的列间距。运行工程，效果如图 3-56 所示。

UICollectionViewDelegateFlowLayout 协议中定义了许多灵活布局的相关方法，例如下面的代理方法中可以灵活设置每个 item 的尺寸大小。

```
-(CGSize)collectionView:(UICollectionView *)collectionView layout:(UICollectionViewLayout *)collectionViewLayout sizeForItemAtIndexPath:(NSIndexPath *)indexPath{
    if (indexPath.row%2==0) {
        return CGSizeMake(50, 50);
    }else{
        return CGSizeMake(100, 100);
    }
}
```

上面代码将单数位置 item 的尺寸设置为 50*50，双数位置 item 尺寸设置为 100*100，运行工程，效果如图 3-57 所示。

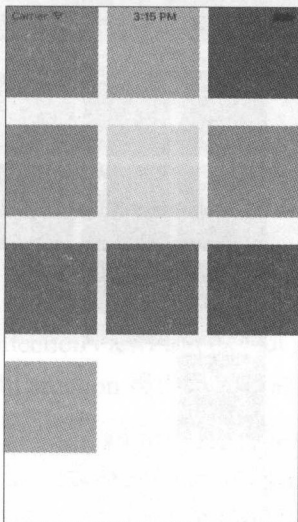


图 3-56 自定义 UICollectionView 中行列间距

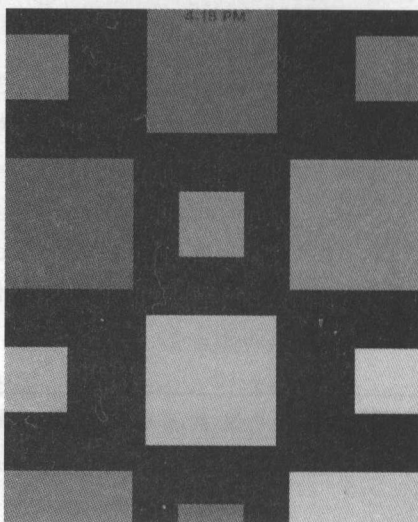


图 3-57 灵活设置 UICollectionView 中每个 item 的尺寸

3.6.4 自定义 UICollectionViewFlowLayout 进行参差瀑布流布局

在很多应用程序中会有瀑布流效果，即分成两列或者多列进行数据的展示，每条数据的载体 item 高度也随数据多少不同而显示的参差不齐。使用原生的 UICollectionViewFlowLayout 类进行布局设置很难实现这样的效果，开发者可以自定义一个它的子类来实现瀑布流式的效果。

首先创建一个布局类，取名 MyLayout，使其继承自 UICollectionViewFlowLayout 类，如图 3-58 所示。

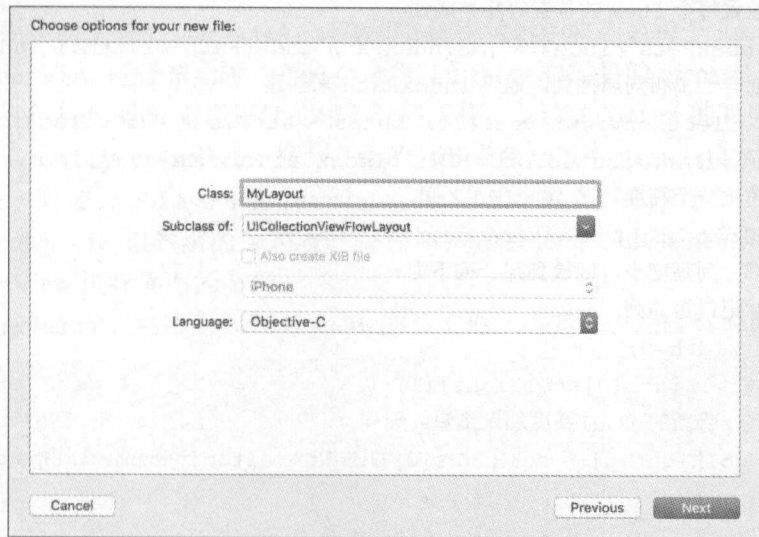


图 3-58 创建一个自定义的布局类

在 MyLayout.h 文件中新增一个 itemCount 属性，用于设置要布局多少个 item。

```
@interface MyLayout : UICollectionViewFlowLayout
@property(nonatomic,assign)int itemCount;
@end
```

在 MyLayout.m 文件中编写如下代码。

```
@implementation MyLayout
{
    //这个数组就是我们自定义的布局配置数组
    NSMutableArray * _attributeArray;
}
//数组的相关设置在这个方法中
//布局前的准备会调用这个方法
-(void)prepareLayout{
    _attributeArray = [[NSMutableArray alloc] init];
    [super prepareLayout];
    //为演示方便，我们设置为静态的 2 列
    //计算每一个 item 的宽度
```

```

float WIDTH = ([UIScreen mainScreen].bounds.size.width-self.sectionInset.left-self.sectionInset.right-self.minimumInteritemSpacing)/2;
//定义数组保存每一列的高度
//这个数组的主要作用是保存每一列的总高度，这样在布局时，我们可以始终将下一个 Item 放在最短的列下面
CGFloat colHight[2]={self.sectionInset.top,self.sectionInset.bottom};
//itemCount 是外界传进来的 item 的个数，遍历来设置每一个 item 的布局
for (int i=0; i<_itemCount; i++) {
    //设置每个 item 的位置等相关属性
    NSIndexPath *index = [NSIndexPath indexPathForItem:i inSection:0];
    //创建一个布局属性类，通过 indexPath 来创建
    UICollectionViewLayoutAttributes * attris = [UICollectionViewLayoutAttributes layoutAttributesForCellWithIndexPath:index];
    //随机一个高度，在 40~190 之间
    CGFloat hight = arc4random()%150+40;
    //哪一列高度小，则放到那一列下面
    //标记最短的列
    int width=0;
    if (colHight[0]<colHight[1]) {
        //将新的 item 高度加入到短的一列
        colHight[0] = colHight[0]+hight+self.minimumLineSpacing;
        width=0;
    }else{
        colHight[1] = colHight[1]+hight+self.minimumLineSpacing;
        width=1;
    }

    //设置 item 的位置
    attris.frame = CGRectMake(self.sectionInset.left+(self.minimumInteritemSpacing+WIDTH)*width, colHight[width]-hight-self.minimumLineSpacing, WIDTH, hight);
    [_attributeArray addObject:attris];
}

//设置 itemSize 来确保滑动范围的正确，这里是通过将所有的 item 高度平均化，计算出来的 (以最高的列位标准)
if (colHight[0]>colHight[1]) {
    self.itemSize = CGSizeMake(WIDTH, (colHight[0]-self.sectionInset.top)*2/_itemCount-self.minimumLineSpacing);
}else{
    self.itemSize = CGSizeMake(WIDTH, (colHight[1]-self.sectionInset.top)*2/_itemCount-self.minimumLineSpacing);
}

```



```

}
//这个方法中返回我们的布局数组
-(NSArray<UICollectionViewLayoutAttributes *> *)layoutAttributesForElementsInRect:(CGRect)rect{
    return _attributeArray;
}
@end

```

上面的代码先声明了一个 `_attributeArray` 数组, 这个数组中将存放每个 item 的布局相关信息。在 `UICollectionView` 进行布局时, 首先会调用其 `Layout` 布局类的 `prepareLayout` 方法, 在这个方法中可以进行每个 item 布局属性的相关计算操作, 具体每个 item 的布局属性实际是保存在 `UICollectionViewLayoutAttributes` 类对象中的, 其中包括每个 item 的尺寸、位置等信息, 其与每个 item 一一对应。在 `Layout` 的 `prepareLayout` 方法中准备好所有 item 的 `UICollectionViewLayoutAttributes` 后, 以数组的形式通过调用 `layoutAttributesForElementsInRect:` 方法来返回给 `UICollectionView` 进行界面的布局。

在 `ViewController.m` 文件中引入 `MyLayout` 的头文件, 将其中方法改写成如下所示:

```

- (void)viewDidLoad {
    [super viewDidLoad];
    MyLayout * layout = [[MyLayout alloc] init];
    layout.itemCount = 20;
    UICollectionView * collectionView = [[UICollectionView alloc] initWithFrame:self.view.frame collectionViewLayout:layout];
    collectionView.backgroundColor = [UIColor whiteColor];
    [collectionView registerClass:[UICollectionViewCell class] forCellWithReuseIdentifier:@"cellID"];
    collectionView.delegate=self;
    collectionView.dataSource=self;
    [self.view addSubview:collectionView];
}

- (NSInteger)numberOfSectionsInCollectionView: (UICollectionView *)collectionView{
    return 1;
}

- (NSInteger)collectionView: (UICollectionView *)collectionView numberOfItemsInSection: (NSInteger) section{
    return 20;
}

- (UICollectionViewCell *)collectionView: (UICollectionView *)collectionView cellForItemAtIndexPath: (NSIndexPath *)indexPath{
    UICollectionViewCell * cell = [collectionView dequeueReusableCellWithReuseIdentifier:@"cellID" forIndexPath:indexPath];
}

```



```

        cell.backgroundColor = [UIColor colorWithRed:arc4random()%255/255.0 gr
een:arc4random()%255/255.0 blue:arc4random()%255/255.0 alpha:1];
        return cell;
    }

```

上面代码将布局类换成了自定义的 MyLayout 类，运行工程，效果如图 3-59 所示。

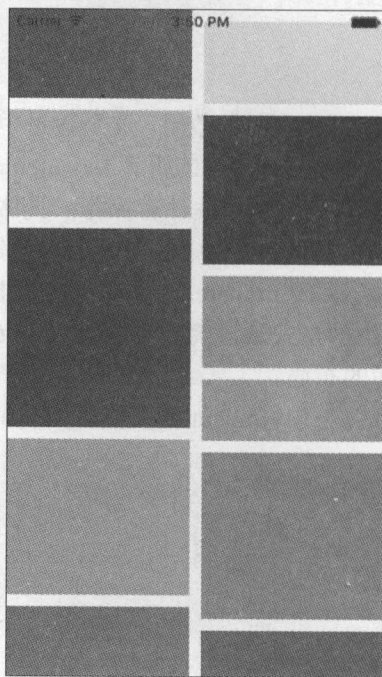


图 3-59 参差的瀑布流布局

3.6.5 使用 UICollectionView 进行圆环布局

UICollectionView 的布局原理是采用 Layout 类进行每个 item 的布局信息配置，具体的配置信息是由 UICollectionViewLayoutAttributes 存储的。明白了这个机制，开发者可以发挥想象，实现更加炫酷复杂的布局效果。使用 Xcode 创建一个名为 UICollectionViewTestTwo 的工程，创建一个新的文件，命名为 MyLayout，使其继承于 UICollectionViewLayout。在 MyLayout.h 中添加一个 itemCount 属性，标记 item 个数。

```

@interface MyLayout : UICollectionViewLayout
//这个 int 值存储有多少个 item
@property(nonatomic,assign)int itemCount;
@end

```

在 MyLayout.m 中编写如下代码：

```

@implementation MyLayout
{
    NSMutableArray * _attributeAttay;
}

```

```

}

-(void)prepareLayout{
    [super prepareLayout];
    //获取 item 的个数
    _itemCount = (int)[self.collectionView numberOfItemsInSection:0];
    _attributeAttay = [[NSMutableArray alloc] init];
    //先设定大圆的半径 取长和宽最短的
    CGFloat radius = MIN(self.collectionView.frame.size.width, self.collectionView.frame.size.height)/2;
    //计算圆心位置
    CGPoint center = CGPointMake(self.collectionView.frame.size.width/2, self.collectionView.frame.size.height/2);
    //设置每个 item 的大小为 50*50 则半径为 25
    for (int i=0; i<_itemCount; i++) {
        UICollectionViewLayoutAttributes * attris = [UICollectionViewLayoutAttributes layoutAttributesForCellWithIndexPath:[NSIndexPath indexPathForItem:i inSection:0]];
        //设置 item 大小
        attris.size = CGSizeMake(50, 50);
        //计算每个 item 的圆心位置
        /*
        .
        . .
        . . r
        . .
        . . . . .
        */
        //计算每个 item 中心的坐标
        //算出的 x y 值还要减去 item 自身的半径大小
        float x = center.x+cosf(2*M_PI/_itemCount*i)*(radius-25);
        float y = center.y+sinf(2*M_PI/_itemCount*i)*(radius-25);

        attris.center = CGPointMake(x, y);
        [_attributeAttay addObject:attris];
    }

}

//设置内容区域的大小
-(CGSize)collectionViewContentSize{
    return self.collectionView.frame.size;
}

```

```

//返回设置数组
- (NSArray<UICollectionViewLayoutAttributes *> *)layoutAttributesForElementsInRect: (CGRect) rect {
    return _attributeAttay;
}
@end

```

上面代码通过一些简单的几何计算, 设置每个 item 的布局位置。collectionViewContentSize 方法将返回一个 UICollectionView 可以滑动的范围尺寸, 由于本节创建的布局模型为圆环布局, 这里返回 UICollectionVie 的原始大小即可。

在 ViewController.m 文件中导入 MyLayout.h 文件, 编写如下代码:

```

@interface ViewController () <UICollectionViewDataSource, UICollectionViewDelegate>
@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    MyLayout * layout = [[MyLayout alloc] init];
    UICollectionView * collect = [[UICollectionView alloc] initWithFrame:CGRectMake(0, 0, 320, 400) collectionViewLayout:layout];
    collect.delegate=self;
    collect.dataSource=self;

    [collect registerClass:[UICollectionViewCell class] forCellWithReuseIdentifier:@"cellid"];
    [self.view addSubview:collect];
}

- (NSInteger)numberOfSectionsInCollectionView: (UICollectionView *) collectionView {
    return 1;
}

- (NSInteger)collectionView: (UICollectionView *)collectionView numberOfItemsInSection: (NSInteger) section {
    return 10;
}

- (UICollectionViewCell *)collectionView: (UICollectionView *)collectionView cellForItemAtIndexPath: (NSIndexPath *)indexPath {
    UICollectionViewCell * cell = [collectionView dequeueReusableCellWithReuseIdentifier:@"cellid" forIndexPath:indexPath];
    cell.layer.masksToBounds = YES;
}

```



```

cell.layer.cornerRadius = 25;
cell.backgroundColor = [UIColor colorWithRed:arc4random()%255/255.0 gr
een:arc4random()%255/255.0 blue:arc4random()%255/255.0 alpha:1];
return cell;
}
@end

```

运行工程，效果如图 3-60 所示。

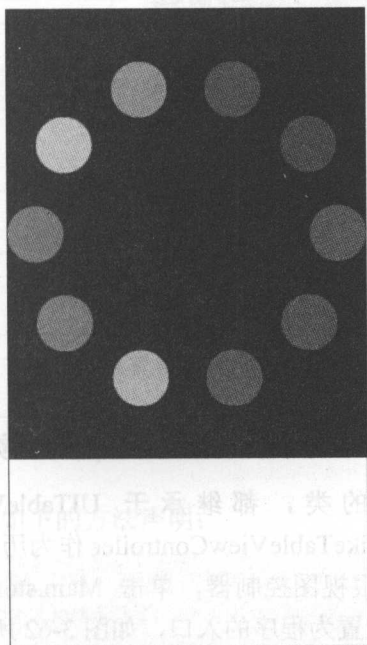


图 3-60 使用 UICollectionView 进行圆环布局

3.7 实战：开发一款手机网页浏览器

本节将编写一款相对完整的网页浏览器应用，一款网页浏览器，应该支持以下功能：

- (1) 用户输入网址进行网页的跳转。
- (2) 用户浏览网页的历史记录并且支持清空历史记录。
- (3) 用户对某些网页的收藏并支持逐个删除收藏的数据。
- (4) 支持网页的前进和后退。
- (5) 在用户浏览网页时，支持一些方便的手势增加用户体验。

通过分析上面的需求列表，这款网页浏览器应用至少应该有 3 个视图控制器，一个核心网页浏览界面，一个历史记录界面和一个收藏界面。核心网页浏览界面可以以 UIWebView 为主进行搭建，历史记录界面和收藏界面可以通过 UITableView 来设计。

3.7.1 网页浏览器工程的搭建

使用 Xcode 创建一个名为 MySafari 的工程，为了支持 https 类型的网页链接，首先在工程配置文件 Info.plist 中添加如图 3-61 所示的键值对。

Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
▼ App Transport Security Settin...	Dictionary	(1 item)
Allow Arbitrary Loads	Boolean	YES
Localization native development re...	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	畅游
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
► Required device capabilities	Array	(1 item)
► Supported interface orientations	Array	(3 items)

图 3-61 在工程配置文件中添加支持 http 类型网页链接的键值对

在工程中创建两个新的类，都继承于 UITableViewController，分别取名为 HistoryTableViewController 和 LikeTableViewController 作为历史记录界面和收藏界面。

将项目工程改为以导航为根视图控制器，单击 Main.storyboard 文件，向其中拖入一个 Navigation Controller，并将其设置为程序的入口，如图 3-62 所示。

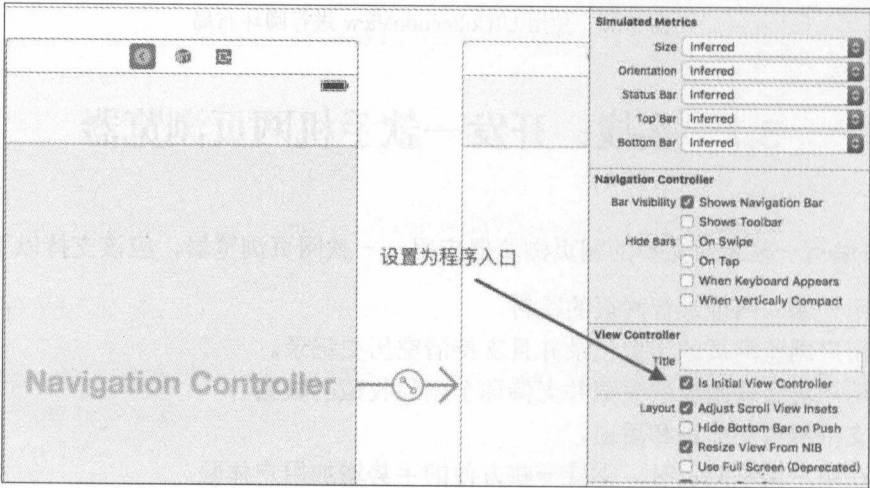


图 3-62 将导航控制器设置为程序入口

将导航控制器自带的 rootViewController 删掉，单击导航控制器，按住 control 键，将鼠标移动至 ViewController 上松开 control 键，单击 root view controller，将 ViewController 设置为导航控制器的根视图控制器，如图 3-63 所示。

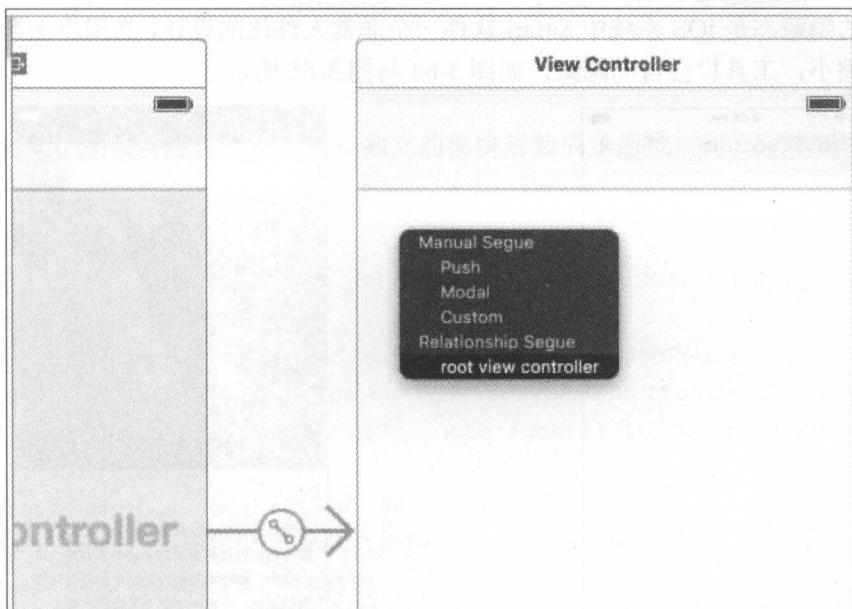


图 3-63 设置导航的根视图控制器

3.7.2 核心网页视图的设计

在 ViewController.h 中添加如下的方法声明：

```
@interface ViewController : UIViewController
-(void)loadURL:(NSString *)urlStr;
@end
```

这个方法用于提供给外界一个加载网页地址的接口。在 ViewController.m 中添加遵守协议的代码并声明一些变量如下：

```
@interface ViewController ()<UIWebViewDelegate, UIGestureRecognizerDelegat
e>
{
    UIWebView * _webView;
    UITextField * _searchBar;
    BOOL _isUp;
    UILabel * _titleLabel;
    UISwipeGestureRecognizer * _upSwipe;
    UISwipeGestureRecognizer * _downSwipe;
}
@end
```

上面声明的变量中，_webView 是核心网页视图；_searchBar 是地址栏；_isUp 用于标记导航栏和工具栏是否处于隐藏状态；_titleLabel 用于显示当前网页的网址；_upSwipe 是上滑手势；_downSwipe 是下滑手势。

导航栏的隐藏态是 iOS 系统中 Safari 软件一个非常人性化的设计，当用户上滑网页时，导航栏会自动缩小，工具栏会自动隐藏，如图 3-64 与图 3-65 所示。

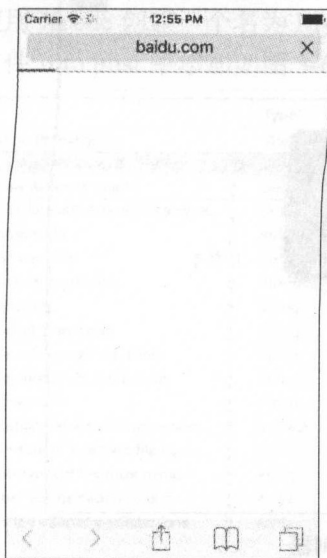


图 3-64 正常状态的导航栏和工具栏



图 3-65 隐藏状态的导航栏和工具栏

在 `ViewController.m` 文件的 `viewDidLoad` 方法中进行一些创建于初始化的操作，代码如下：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    _webView = [[UIWebView alloc] initWithFrame:CGRectMake(0, 64, self.view.
frame.size.width, self.view.frame.size.height-64)];
    _webView.scrollView.bounces=NO;
    _webView.delegate=self;
    _isUp = NO;
    _titleLabel = [[UILabel alloc] initWithFrame:CGRectMake(0, 0, self.view.
frame.size.width-40, 20)];
    _titleLabel.backgroundColor = [UIColor clearColor];
    _titleLabel.font = [UIFont systemFontOfSize:14];
    _titleLabel.textAlignment = NSTextAlignmentCenter;
    //默认加载百度
    NSURL * url = [NSURL URLWithString:[NSString stringWithFormat:@"http://
/www.baidu.com"]];
    NSURLRequest * request = [NSURLRequest requestWithURL:url];
    [_webView loadRequest:request];
    [self.view addSubview:_webView];
    //对导航栏进行设置
    [self creatSearchBar];
    //创建手势
    [self creatGesture];
}
```

```
//创建工具栏
[self creatToolBar];
}
```

上面代码默认加载百度首页，将一些独立的模块封装成了函数，creatSearchBar 方法的实现如下所示：

```
-(void)creatSearchBar{
    _searchBar = [[UITextField alloc] initWithFrame:CGRectMake(0, 0, self.view.frame.size.width-40, 30)];
    _searchBar.borderStyle = UITextBorderStyleRoundedRect;
    UIButton * goBtn = [UIButton buttonWithType:UIButtonTypeSystem];
    [goBtn addTarget:self action:@selector(goWeb) forControlEvents:UIControlEventTouchUpInside];
    goBtn.frame=CGRectMake(0, 0, 30, 30);
    [goBtn setTitle:@"GO" forState:UIControlStateNormal];
    _searchBar.rightView = goBtn;
    _searchBar.rightViewMode = UITextFieldViewModeAlways;
    _searchBar.placeholder = @"请输入网址";
    self.navigationItem.titleView = _searchBar;
}
```

creatSearchBar 方法中对地址栏进行了初始化和相关设置，并将其设置为了导航栏的标题视图，在地址栏中添加了一个 GO 按钮，当用户单击这个按钮时，将进行网页的跳转，goWeb 方法的实现如下所示：

```
-(void)goWeb{
    if (_searchBar.text.length>0) {
        NSURL * url = [NSURL URLWithString:[NSString stringWithFormat:@"http://%@", _searchBar.text]];
        NSURLRequest * request = [NSURLRequest requestWithURL:url];
        [_webView loadRequest:request];
    }else{
        UIAlertController * alert = [UIAlertController alertControllerWithTitle:@"温馨提示" message:@"输入的网址不能为空" preferredStyle:UIAlertControllerStyleAlert];
        UIAlertAction * action = [UIAlertAction actionWithTitle:@"好的" style:UIAlertActionStyleDefault handler:nil];
        [alert addAction:action];
        [self presentViewController:alert animated:YES completion:nil];
        return;
    }
}
```

在 goWeb 方法中进行了一个逻辑判断，如果地址栏中没有文字，则会弹出一个警告框提示用户输入的网址不能为空，如果地址栏中有文字，则使 _webView 加载新的网页。

创建手势的 creatGesture 方法实现如下所示：

```
-(void)creatGesture{
    _upSwipe = [[UISwipeGestureRecognizer alloc] initWithTarget:self action:
@selector(upSwipe)];
    _upSwipe.delegate=self;
    _upSwipe.direction = UISwipeGestureRecognizerDirectionUp;
    [_webView addGestureRecognizer:_upSwipe];

    _downSwipe = [[UISwipeGestureRecognizer alloc] initWithTarget:self action:
@selector(downSwipe)];
    _downSwipe.delegate=self;
    _downSwipe.direction = UISwipeGestureRecognizerDirectionDown;
    [_webView addGestureRecognizer:_downSwipe];
}
```

UISwipeGestureRecognizer 类是专门用来处理滑动手势的类，读者在此只需关注其应用，不需要深究这个类本身。UISwipeGestureRecognizer 类的 direction 属性设置滑动的方向，可选枚举如下所示：

```
typedef NS_OPTIONS(NSUInteger, UISwipeGestureRecognizerDirection) {
    UISwipeGestureRecognizerDirectionRight = 1 << 0, //右滑手势
    UISwipeGestureRecognizerDirectionLeft = 1 << 1, //左滑手势
    UISwipeGestureRecognizerDirectionUp = 1 << 2, //上滑手势
    UISwipeGestureRecognizerDirectionDown = 1 << 3, //下滑手势
};
```

UISwipeGestureRecognizerDirection 枚举是支持进行位运算的，例如要同时支持左滑和右滑手势，可以使用如下代码：

```
_upSwipe.direction = UISwipeGestureRecognizerDirectionRight|UISwipeGestureRecognizerDirectionLeft;
```

上滑手势的触发方法 upSwipe 实现如下：

```
-(void)upSwipe{
    if (!_isUp) {
        return;
    }
    self.navigationItem.titleView=nil;
    _webView.frame=CGRectMake(0, 40, self.view.frame.size.width, self.view.frame.size.height-40);
    [UIView animateWithDuration:0.3 animations:^(
```



```

        self.navigationController.navigationBar.frame=CGRectMake(0, 0, self.navigationController.navigationBar.frame.size.width, 40);
        [self.navigationController.navigationBar setTitleVerticalPositionAdjustment:7 forBarMetrics:UIBarMetricsDefault];

        } completion:^(BOOL finished) {
            self.navigationItem.titleView = _titleLabel;

        }];
        [self.navigationController setToolbarHidden:YES animated:YES];
        _isUp=YES;
    }

```

在这个方法中先判断导航栏和工具栏是否处于隐藏状态,如果是,则不执行任何逻辑,如果不是,则进行导航栏工具栏尺寸的重设和相关属性的修改。UIView 的类方法 `animateWithDuration:animations:completion:` 方法是 UIView 层用于处理渐变动画的方法,这个方法中第 1 个参数设置动画执行的时间,第 2 个参数是一个 block 函数块,将要执行动画的属性操作代码放入其中,第 3 个参数是动画结束后的回调。

下滑手势的触发方法 `downSwipe` 实现如下:

```

-(void)downSwipe{
    if (_webView.scrollView.contentOffset.y==0&&_isUp) {
        self.navigationItem.titleView=nil;
        _webView.frame=CGRectMake(0, 64, self.view.frame.size.width, self.view.frame.size.height-64);
        [UIView animateWithDuration:0.3 animations:^(
            self.navigationController.navigationBar.frame=CGRectMake(0, 0, self.navigationController.navigationBar.frame.size.width, 64);
            [self.navigationController.navigationBar setTitleVerticalPositionAdjustment:0 forBarMetrics:UIBarMetricsDefault];
        ) completion:^(BOOL finished) {
            self.navigationItem.titleView = _searchBar;
        }];
        [self.navigationController setToolbarHidden:NO animated:YES];
        _isUp=NO;
    }
}

```

`downSwipe` 方法与 `upSwipe` 方法类似,只是将导航栏和工具栏从隐藏状态还原回正常状态,通过动画来展示这个过程。

创建工具栏的 `creatToolBar` 方法实现如下:

```

-(void)creatToolBar{
    self.navigationController.toolbarHidden=NO;
}

```

```

UIBarButtonItem * itemHistory = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemCompose target:self action:@selector(goHistory)];

UIBarButtonItem * itemLike = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemBookmarks target:self action:@selector(goLike)];

UIBarButtonItem * itemBack = [[UIBarButtonItem alloc] initWithTitle:@"back" style:UIBarButtonItemStylePlain target:self action:@selector(goBack)];

UIBarButtonItem * itemForward = [[UIBarButtonItem alloc] initWithTitle:@"forward" style:UIBarButtonItemStylePlain target:self action:@selector(goForward)];

UIBarButtonItem * emptyItem = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace target:nil action:nil];

UIBarButtonItem * emptyItem2 = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace target:nil action:nil];

UIBarButtonItem * emptyItem3 = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace target:nil action:nil];

self.toolbarItems = @[itemHistory, emptyItem, itemLike, emptyItem2, itemBack, emptyItem3, itemForward];
}

```

creatToolBar 方法中创建了 4 个功能按钮，分别为历史记录按钮、收藏按钮、前进按钮和后退按钮。emptyItem、emptyItem2 和 emptyItem3 按钮没有实际的用途，只是为了布局进行占位。goHistory 和 goLike 方法分别进行历史记录界面和收藏界面的跳转操作。

先在 ViewController.m 中引入如下头文件：

```

#import "HistoryTableViewController.h"
#import "LikeTableViewController.h"

```

goHistory 方法实现如下：

```

-(void)goHistory{
    HistoryTableViewController * controller = [[HistoryTableViewController alloc] init];
    [self.navigationController pushViewController:controller animated:YES];
}

```

goHistory 方法中创建了一个 HistoryTableViewController 的对象，并通过导航将其 push 出来。goLike 方法实现如下：

```

-(void)goLike{
    UIAlertController * alert = [UIAlertController alertControllerWithTitle:@"温馨提示" message:@"请选择您要进行的操作" preferredStyle:UIAlertControllerStyleActionSheet];
}

```

```

    UIAlertAction * action = [UIAlertAction actionWithTitle:@"添加收藏" style:UIAlertActionStyleDefault handler:^(UIAlertAction * _Nonnull action) {
        NSArray * array = [[NSUserDefaults standardUserDefaults] valueForKey:@"Like"];
        if (!array) {
            array = [[NSArray alloc] init];
        }
        NSMutableArray * newArray = [[NSMutableArray alloc] initWithArray:array];
        [newArray addObject:_webView.request.URL.absoluteString];
        [[NSUserDefaults standardUserDefaults] setValue:newArray forKey:@"Like"];
        [[NSUserDefaults standardUserDefaults] synchronize];
    }];
    UIAlertAction * action2 = [UIAlertAction actionWithTitle:@"查看收藏夹" style:UIAlertActionStyleDefault handler:^(UIAlertAction * _Nonnull action) {
        LikeTableViewController * controller = [[LikeTableViewController alloc] init];
        [self.navigationController pushViewController:controller animated:YES];
    }];
    UIAlertAction * action3 = [UIAlertAction actionWithTitle:@"取消" style:UIAlertActionStyleCancel handler:nil];
    [alert addAction:action];
    [alert addAction:action2];
    [alert addAction:action3];
    [self presentViewController:alert animated:YES completion:nil];
}

```

用户单击收藏按钮后，先弹出一个活动列表，其中有添加收藏和查看收藏夹两种操作。单击添加收藏后，将当前的网址添加进数组并通过 `NSUserDefaults` 进行本地化保存。`NSUserDefaults` 是 iOS 中常用的简单数据本地化的方式之一，其原理是将常用的数据类型存储为 plist 文件。当用户单击查看收藏夹后，将进行 `LikeTableViewController` 的创建并将其 push 出来。

`goBack` 和 `goForward` 方法的实现十分简单，直接调用 `UIWebView` 中的接口即可，代码如下：

```

-(void)goBack{
    if ([_webView canGoBack]) {
        [_webView goBack];
    }
}
-(void)goForward{
    if ([_webView canGoForward]) {

```



```

        [_webView goForward];
    }
}

```



提示

在调用 UIWebView 的 goBack 和 goForward 方法前都进行了可用性检查。

在 ViewController.m 中还需要实现为外界准备的加载网页的方法 loadURL 和一些代理方法，loadURL 方法的实现如下：

```

-(void)loadURL:(NSString *)urlStr{
    NSURL * url = [NSURL URLWithString:[NSString stringWithFormat:@"%@",urlStr]];
    NSURLRequest * request = [NSURLRequest requestWithURL:url];
    [_webView loadRequest:request];
}

```

UIWebViewDelegate 的相关方法实现如下：

```

-(void)webViewDidFinishLoad:(UIWebView *)webView{
    _titleLabel.text = webView.request.URL.absoluteString;
    NSArray * array = [[NSUserDefaults standardUserDefaults] valueForKey:@"History"];
    if (!array) {
        array = [[NSArray alloc] init];
    }
    NSMutableArray * newArray = [[NSMutableArray alloc] initWithArray:array];
    [newArray addObject:_titleLabel.text];
    [[NSUserDefaults standardUserDefaults] setValue:newArray forKey:@"History"];
    [[NSUserDefaults standardUserDefaults] synchronize];
}

```

当 UIWebView 加载完成后，将加载的网址通过 NSUserDefaults 写入本地进行保存。

UIGestureRecognizerDelegate 的相关方法实现如下：

```

- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer shouldRecognizeSimultaneouslyWithGestureRecognizer:(UIGestureRecognizer *)otherGestureRecognizer
{
    if (gestureRecognizer == _upSwipe || gestureRecognizer == _downSwipe)
    {
        return YES;
    }
    return NO;
}

```

由于 UIWebView 自带滑动手势, 如果开发者不做处理, UIWebView 的滑动手势和开发者添加的滑动手势将会冲突, `gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:` 方法用于处理多手势共同触发的情况。

至此, 手机网页浏览器的核心功能已经实现完成, 运行工程, 效果如图 3-66 与图 3-67 所示。



图 3-66 网页浏览器示例图 1



图 3-67 网页浏览器示例图 2

扩展:

NSUserDefaults 采用单例的设计模式, 在项目工程的任何地方都可以方便地进行数据存取和统一管理。NSUserDefaults 的 `standardUserDefaults` 用于获取单例对象, `setValue:forKey` 用于数据的存入, `synchronize` 方法用于将存入的数据同步到磁盘存入本地, `valueForKey:` 方法用于获取存储的本地数据。

3.7.3 历史记录界面的设计

HistoryTableViewController 继承于 UITableViewController。UITableViewController 是 UIViewController 和 UITableView 的结合, 在使用 UITableViewController 时不需要再编写设置代理与遵守协议的方法, 直接在其中实现相应的方法即可。在 HistoryTableViewController.m 文件中声明一个数组作为历史记录的数据源, 代码如下:

```
@interface HistoryTableViewController ()
{
    NSArray * _dataArray;
}
@end
```

在 viewDidLoad 中添加如下代码:

```

- (void)viewDidLoad {
    [super viewDidLoad];
    _dataArray = [[NSUserDefaults standardUserDefaults] valueForKey:@"History"];
    UIBarButtonItem * item = [[UIBarButtonItem alloc] initWithTitle:@"Delete All" style:UIBarButtonItemStylePlain target:self action:@selector(deleteAll)];
    self.navigationItem.rightBarButtonItem=item;
}

```

上面代码将网页浏览的历史记录读取为数组，并创建了一个导航栏右侧按钮用于清空历史数据。deleteAll 方法的实现如下：

```

- (void)deleteAll{
    [[NSUserDefaults standardUserDefaults] setValue:[NSArray alloc] initWith:@"History"];
    [[NSUserDefaults standardUserDefaults] synchronize];
    _dataArray = @[];
    [self.tableView reloadData];
}

```

在 deleteAll 方法中执行了两步操作，第 1 步将本地的记录数据清除，第 2 步将数据源清空并重新刷新表格视图。

在 HistoryTableViewController.m 中实现 UITableView 的数据源协议和代理协议的相关方法，如下所示：

```

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:(NSInteger)section {
    return _dataArray.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"cellID"];
    if (cell==nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"cellID"];
    }
    cell.textLabel.text = _dataArray[indexPath.row];
    return cell;
}

```



```

-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath{
    [(ViewController *)self.navigationController.viewControllers.firstObject loadURL:_dataArray[indexPath.row]];
    [self.navigationController popViewControllerAnimated:YES];
}

```

上面的方法将 UITableView 表格设置为 1 个分区，行数与数据源中数据个数相对应，当用户单击某一行时，通过导航控制器的 viewControllers 属性获取到导航的根视图控制器。导航的根视图控制器实际上就是核心网页视图控制器 ViewController，让其加载用户单击行的网址数据并使界面弹回核心网页浏览界面。



提示

通过导航间接获取到的 ViewController 对象需要进行强转操作，并且要执行其中的 loadURL:方法,需要在 HistoryTableViewController.m 中引入的头文件如下:

```
#import "ViewController.h"
```

还有一个小细节需要处理，当用户进入历史记录或者收藏界面后，工具栏应该被隐藏，可以在视图控制器的生命周期函数中进行操作，代码如下所示。

```

-(void)viewWillAppear:(BOOL)animated{
    [super viewWillAppear:animated];
    self.navigationController.toolbarHidden=YES;
}

-(void)viewWillDisappear:(BOOL)animated{
    [super viewWillDisappear:YES];
    self.navigationController.toolbarHidden = NO;
}

```

运行工程，单击历史记录按钮，效果如图 3-68 所示。

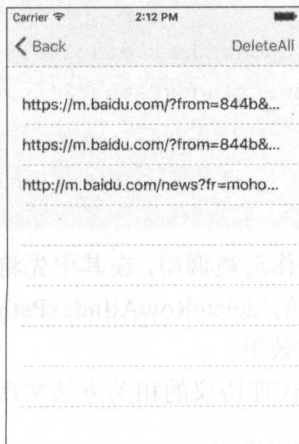


图 3-68 历史记录界面

3.7.4 收藏界面的设计

LikeTableViewController 的设计与 HistoryTableViewController 类似，但也有不同之处，LikeTableViewController 中需要支持逐个数据的删除操作。因此，将 LikeTableViewController 中的数据源声明成可变的数组，代码如下：

```
@interface LikeTableViewController ()
{
    NSMutableArray * _dataArray;
}
@end
```

在 viewDidLoad 方法中添加如下代码：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    _dataArray = [[NSMutableArray alloc] initWithArray:[NSUserDefaults standardUserDefaults] valueForKey:@"Like"]];
    self.navigationItem.rightBarButtonItem = self.editButtonItem;
}
```

editButtonItem 是 UITableViewController 自带的导航编辑按钮，开发者只需要实现如下的代理方法即可。

```
- (void)tableView:(UITableView *)tableView commitEditingStyle:(UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)indexPath {
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        // Delete the row from the data source
        [_dataArray removeObjectAtIndex:indexPath.row];
        [[NSUserDefaults standardUserDefaults] setValue:_dataArray forKey:@"Like"];
        [[NSUserDefaults standardUserDefaults] synchronize];
        [tableView deleteRowsAtIndexPaths:@[indexPath] withRowAnimation:UITableViewRowAnimationFade];
    }
}
```

上面的方法在用户单击删除操作后被调用，在其中先将本条数据从数据源中移去，再将其从本地数据中移去，UITableView 的 deleteRowsAtIndexPaths:withRowAnimation: 方法可以将界面上的此行移去并执行一个动画的效果。

UITableView 的数据源协议和代理协议的相关方法实现如下所示。

```

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:(NSInteger)section {
    return _dataArray.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
(NSIndexPath *)indexPath {
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"cellID"];
    if (cell==nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleD
efault reuseIdentifier:@"cellID"];
    }
    cell.textLabel.text = _dataArray[indexPath.row];
    return cell;
}

-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSInde
xPath *)indexPath{
    [(ViewController *)self.navigationController.viewControllers.firstObj
ect loadURL:_dataArray[indexPath.row]];
    [self.navigationController popViewControllerAnimated:YES];
}

```

类似 HistoryTableViewController, 实现 LikeTableViewController 的生命周期函数如下所示:

```

-(void)viewWillAppear:(BOOL)animated{
    [super viewWillAppear:animated];
    self.navigationController.toolbarHidden=YES;
}

-(void)viewWillDisappear:(BOOL)animated{
    [super viewWillDisappear:YES];
    self.navigationController.toolbarHidden = NO;
}

```

运行工程, 收藏夹界面效果如图 3-69 所示。

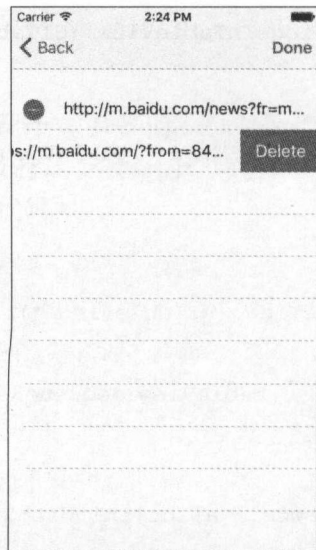


图 3-69 收藏夹界面

3.7.5 启动页面、图标及应用名称的相关优化

几乎所有线上的应用都会有一个启动页面，启动页面主要用于当前应用的介绍、广告、友好地提示用户等界面。为 MySafari 项目添加一个启动页面也十分简单，在 Xcode 的文件导航栏部分找到 LaunchScreen.storyboard 文件，如图 3-70 所示，这个文件就是默认的启动页面，可以在其中进行一些简单的设计，例如向其中添加一个标签，如图 3-71 所示。

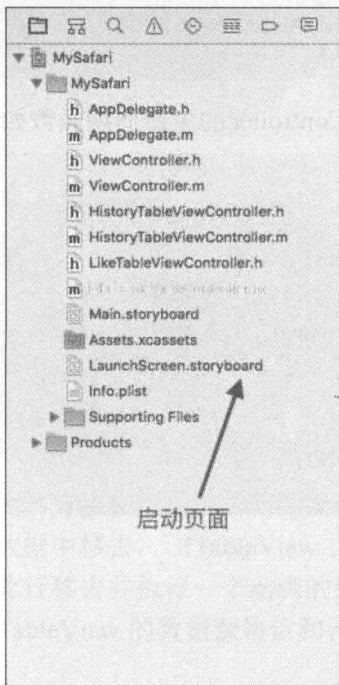


图 3-70 启动页面文件

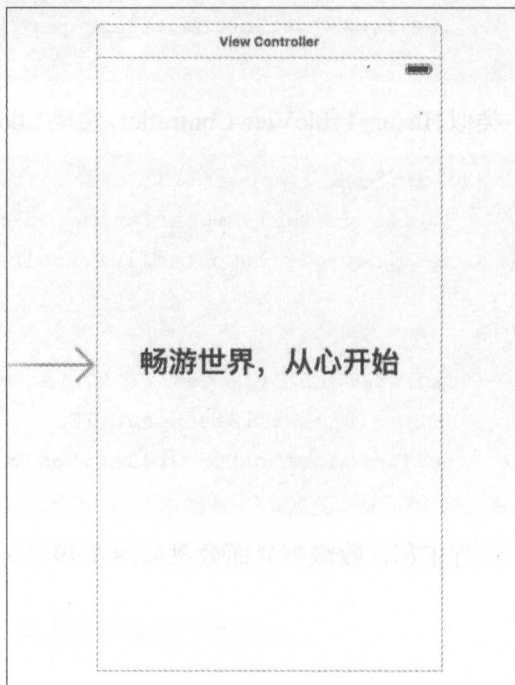


图 3-71 向启动页面中添加一行文字

运行工程，可以看到在应用启动时，会出现闪屏界面。

对于一个应用程序，一个漂亮的图标也是必不可少的，在 Xcode 的文件导航栏中单击 Assets.xcassets 文件，其中自带一个名为 AppIcon 的组，这个组中可以配置各种设备尺寸的图片文件，如图 3-72 所示。

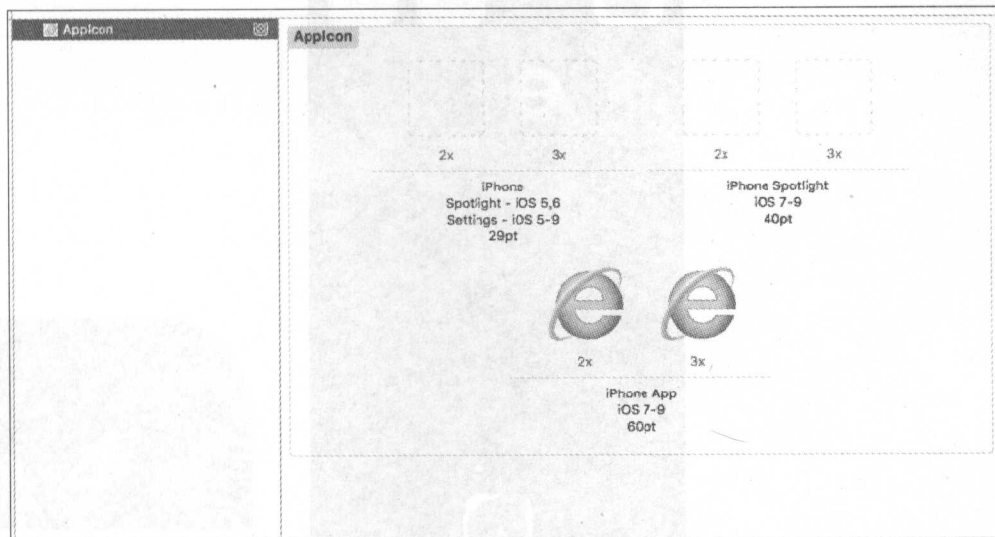


图 3-72 配置应用图标

AppIcon 组中有 3 个尺寸的模型，分别是 29tp, 40tp, 60tp。每个尺寸的模型中可以配置两种尺寸的图片，分别是 2×和 3×，2×需要配置图片的尺寸为模型尺寸乘 2，3×需要配置图片的尺寸为模型尺寸乘 3。例如，60pt 配置的两个图片文件的尺寸分别为 120*120 和 180*180。

在模拟器或者真机上可以看到，应用程序的名称默认是和工程名称一致的，大多数时候，应用的名字往往和工程的名字并不完全一样，可以在 Info.plist 文件中配置 Bundle name 键来设置应用的名称，如图 3-73 所示。

Key	Type	Value
▼ Information Property List	Dictionary	{15 items}
▼ App Transport Security Settings	Dictionary	{1 item}
Allow Arbitrary Loads	Boolean	YES
Localization native development re...	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	畅游
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
Required device capabilities	Array	{1 item}
Supported interface orientations	Array	{3 items}

图 3-73 配置应用名称

再次运行一遍工程，单击 Home 键退回主界面，可以看到配置了图标和名称的网页浏览器应用，如图 3-74 所示。

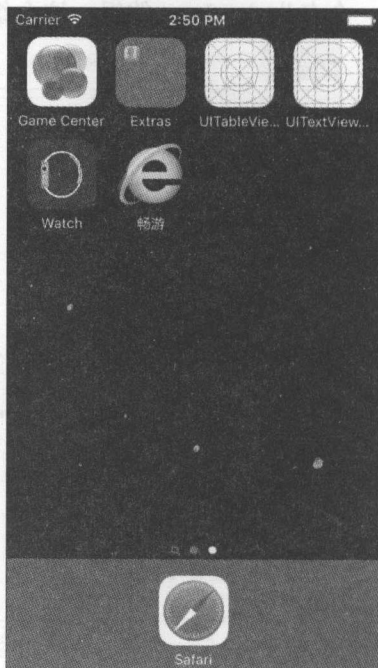


图 3-74 配置了图片和名称的网页浏览器应用



提示

在模拟器上，可以通过单击 `command+shift+h` 键来模拟单击 Home 键。

第 4 章

网络编程

移动网络改变了人们的生活方式。听歌、观看视频、购物、聊天、游戏对战等丰富多彩的应用和游戏在开发时都离不开对网络编程的使用。在 iOS 开发中，除了使用系统原生的网络框架来获取网络数据外，许多优秀的第三方库也使网络编程变得更简洁方便。本章在向读者介绍 iOS 原生网络框架的基础上也将着重使用第三方框架 AFNetworking 来进行网络编程请求的示例。

通过本章的学习，读者能够掌握：

1. 用 iOS 原生框架进行 HTTP/HTTPS 协议的网络请求。
2. 析 JSON 类型的网络数据。
3. 装更易用的网络请求类。
4. 用 CocoaPods 进行工程项目管理。
5. FNetworking 框架的简单使用。
6. 实际应用中使用的网络编程技术。

4.1 使用 NSURLConnection 请求网络数据

NSURLConnection 类是 iOS 中一个处理网络连接的类,其通过一个请求地址 NSURL 和一个具体的请求 NSMutableURLRequest 来进行网络通信。NSURLConnection 支持同步与异步两种方式的网络访问,对于异步的访问,NSURLConnection 支持代理方法和 block 代码块两种方式进行回调处理。

除了即时通讯类应用需要建立长连接保持实时通信外,大多数网络应用都是采用 HTTP/HTTPS 协议进行网络请求,当接收到请求回执后,连接立即断开。网络数据一般由服务端提供,至于数据的来源是文件服务器还是数据库,客户端都不需要关心,客户端只需要专注于与服务端的通信。客户端与服务端通信的过程如图 4-1 所示。

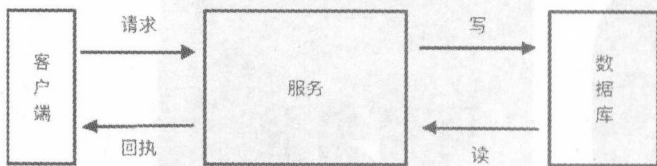


图 4-1 客户端与服务端进行通信

4.1.1 申请一个免费的 API 服务

要学习客户端的网络编程,一个用于学习与测试的接口服务是必不可少的。API 就是应用程序编程接口,通过在自己电脑上搭建本地的服务是可行的,但无疑会增加很大额外的学习成本。幸运的是,互联网上提供了许多免费的简单 API 服务,百度开发的 APIStore 是一个不错的选择,里面提供了许多日常生活中常用的一些 API 服务,并且大部分都是免费的,开发者使用其进行学习、测试和编写有趣的网络小程序都是十分方便的。

通过 apistore.baidu.com 网址打开 APIStore 网站,要使用其中的 API 服务需要先注册成为开发者。首先单击网页上的登录按钮进行用户登录,如果没有百度账号,可以进行免费注册,如图 4-2 所示。

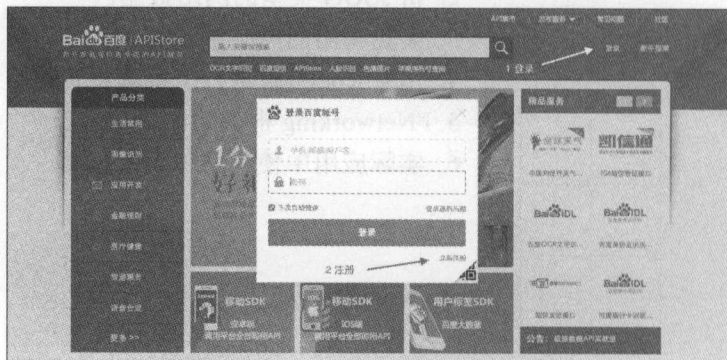


图 4-2 进行 APIStore 网站的登录操作

使用百度账号登录后,需要进入个人中心进行个人信息的完善和手机验证操作,如图4-3所示,手机验证成功后,可以使用 APIStore 提供的免费 API 服务。

图 4-3 完善个人信息和进行手机验证

提示

在个人中心里可以看到 `apikey` 这样一串字符,在调用 APIStore 提供的免费接口服务时这个参数用于身份验证,是必不可少的。读者在学习时,可以使用 `c925fbc1226c37b905a4d1e2a8cbbe99` 进行接口的访问,也可以使用自己通过手机验证的账号生成的 `apikey`。

回到 APIStore 的首页,将费用选项选择为免费,可以看到所有的免费 API 服务列表,如图4-4所示。



图 4-4 获取免费 API 服务列表

在免费的 API 服务列表中选择一个接口服务，例如选择笑话大全服务，里面会提供接口的访问地址、请求方法、请求的参数和多种语言的请求示例，如图 4-5 所示。

接口地址：

http://apis.baidu.com/showapi_open_bus/showapi_joke/joke_text

请求方法：

GET

请求参数(header)：

参数名	类型	必填	参数位置	描述	默认值
apikey	string	是	header	API密钥	您自己的apikey

请求参数(uriParam)：

参数名	类型	必填	参数位置	描述	默认值
page	string	是	uriParam	第几页。每页最多返回20条	1

请求示例：

curl示例

php示例

python示例

java示例

C#示例

ObjectC示例

Swift示例

1

```
curl --get --include 'http://apis.baidu.com/showapi_open_bus/showapi_joke/joke_text?page=1' -H 'apikey:您自己的apikey'
```

图 4-5 API 服务的简单文档说明

要验证接口服务是否已经有效，在 MAC 上可以使用终端工具来进行测试，单击终端工具，如图 4-6 所示。

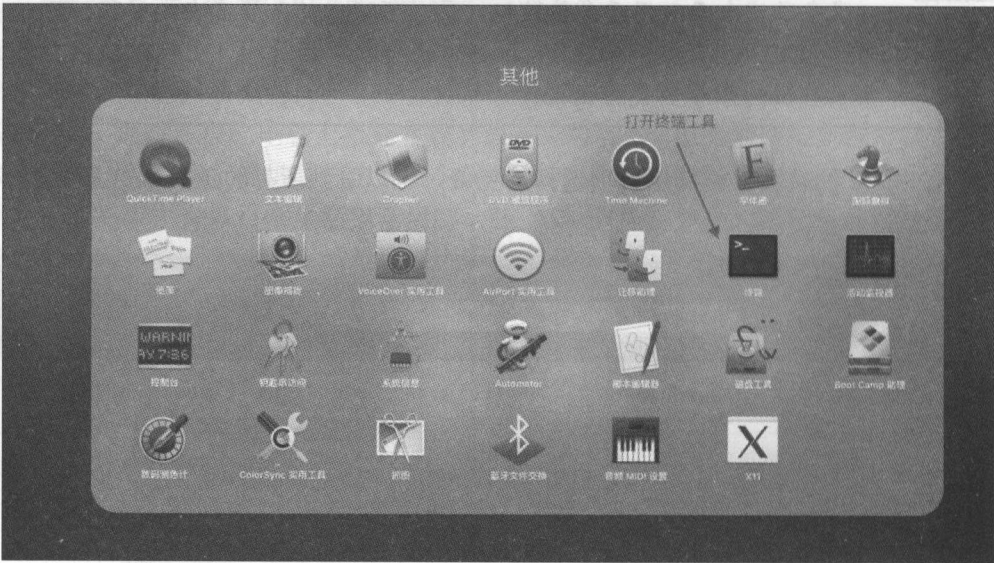


图 4-6 使用终端工具

在终端中输入如下命令：

```
curl --get --include 'http://apis.baidu.com/showapi_open_bus/showapi_joke/joke_text?page=1' -H 'apikey:c925fbc1226c37b905a4d1e2a8cbb99'
```

敲击 return 键，终端返回如图 4-7 所示的数据信息，代表 API 服务可用，接口测试成功。

```
apple — bash — 80x24
Last login: Tue Feb 16 12:02:35 on ttys000
appledeMacBook-Air:~ apple$ curl --get --include 'http://apis.baidu.com/showapi_open_bus/showapi_joke/joke_text?page=1' -H 'apikey:c925fbc1226c37b905a4d1e2a8cbb99'
HTTP/1.1 200 OK
Server: nginx/1.7.10
Date: Tue, 16 Feb 2016 06:17:14 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept-Encoding
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: apikey
Access-Control-Allow-Methods: GET, POST, OPTIONS

{"showapi_res_code":0,"showapi_res_error":"","showapi_res_body":{"allNum":17052,"allPages":853,"contentlist":[{"ct":"2016-02-15 15:30:48.822","text":"老婆：我是你的什么？老公：你是我的红烧肉呀！老婆：是不是你一天不见我就馋得慌？老公：不是，就你这么肥而我却不腻","title":"我是你的什么","type":1}],"ct":"2016-02-15 15:30:4
```

图 4-7 使用终端进行 API 服务测试

4.1.2 使用 NSURLConnection 进行 API 服务数据的获取

在 4.1.1 小节中，实际上是通过终端使用 curl 工具进行了接口数据的请求，在 iOS 开发中，这一过程可以通过 NSURLConnection 类来完成。

使用 Xcode 创建一个名为 NSURLConnectionTest 的工程，为了支持 HTTP 类型协议的请求，在 Info.plist 文件中添加如图 4-8 所示的键值。

Key	Type	Value
Information Property List	Dictionary	(15 items)
App Transport Security Settings	Dictionary	(1 item)
Allow Arbitrary Loads	Boolean	YES
Localization native development re...	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(3 items)

图 4-8 在 Info.plist 文件中进行键值配置

在 ViewController.m 文件的 viewDidLoad 中添加如下代码：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSURL * url = [NSURL URLWithString:@"http://apis.baidu.com/showapi_o
n_bus/showapi_joke/joke_text?page=1"];
    NSMutableURLRequest * request = [NSMutableURLRequest requestWithURL:ur
ll];
```

```

[request setValue:@"c925fbc1226c37b905a4d1e2a8cbb99" forHTTPHeaderField:@"apikey"];
NSData * data = [NSURLConnection sendSynchronousRequest:request returningResponse:nil error:nil];
NSString * dataStr = [[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding];
NSLog(@"%@", dataStr);
}

```

上面代码中，NSURL 创建了请求地址。NSMutableURLRequest 是具体的请求对象，其 setValue:forHTTPHeaderField: 方法设置请求头中的字段。NSURLConnection 类的类方法 sendSynchronousRequest:returningResponse:error: 用于同步进行数据请求，其返回值为 NSData 类型的返回数据。使用 NSString 的 initWithData:encoding: 方法将 NSData 数据读取为字符串。



提示

程序代码的执行分为同步和异步两种方式，同步执行会阻塞当前线程，即只有在当前代码执行完成后程序才会继续向后执行，异步执行在执行时不会阻塞当前线程，程序会继续向后执行。上面的请求方式为同步请求，只有当请求的数据返回后，才会执行后面的转换字符串和打印数据操作。

运行工程，在打印调试时可以看到请求到的数据，如图 4-9 所示。



图 4-9 打印的网络数据



提示

在 iOS 9 之后，NSURLConnection 被 NSURLSession 取代。有关 NSURLConnection 的方法都被弃用，因此在 Xcode 7.0 之上的版本中使用 NSURLConnection 的相关方法会被警告，这里读者不必介意，NSURLConnection 虽被弃用，但 iOS 有向下兼容的特性，其方法仍然有效。

4.1.3 使用 NSURLConnection 进行异步网络请求

大部分应用中有关网络请求的操作都会采用异步的方式，网络数据的传输需要一定的时间，对于数据量大的请求更是如此。如果进行同步请求，在请求期间，程序的界面会被卡死，

使用户完全无法操作，导致用户体验极差。NSURLConnection 同样支持异步请求操作，将 viewDidLoad 中的代码修改如下：

```

- (void)viewDidLoad {
    [super viewDidLoad];
    NSURL * url = [NSURL URLWithString:@"http://apis.baidu.com/showapi_ope
n_bus/showapi_joke/joke_text?page=1"];
    NSMutableURLRequest * request = [NSMutableURLRequest requestWithURL:ur
l];
    [request setValue:@"c925fbc1226c37b905a4d1e2a8cbb99" forHTTPHeaderFi
eld:@"apikey"];
    [NSURLConnection sendAsynchronousRequest:request queue:[NSOperationQu
eue mainQueue] completionHandler:^(NSURLResponse * _Nullable response, NSData
* _Nullable data, NSError * _Nullable connectionError) {
        NSString * dataStr = [[NSString alloc] initWithData:data encoding:NS
UTF8StringEncoding];
        NSLog(@"%@", dataStr);
    }];
    NSLog(@"异步进行网络请求");
}

```

上面代码中，使用 NSURLConnection 类的类方法 sendAsynchronousRequest:queue:completionHandler:方法发起异步的网络请求，这个方法中第1个参数为具体的请求对象，第2个参数设置得到返回数据后在队列中进行 block 中方法的执行，第3个参数为获取返回数据后执行的 block 代码块，这个 block 中传入的 data 参数为请求的返回数据。



提示

[NSOperationQueue mainQueue]可以获取到应用运行的主线程队列，有关 UI 的操作必须放入主线程中执行，否则会出现无法控制的延时。

运行工程，在打印出的数据中可以看出，程序先执行“异步进行网络请求”打印，然后才执行请求返回数据的打印，如图 4-10 所示。

```

17 - (void)viewDidLoad {
18     [super viewDidLoad];
19     NSURL * url = [NSURL URLWithString:@"http://apis.baidu.com/
showapi_open_bus/showapi_joke/joke_text?page=1"];
20     NSMutableURLRequest * request = [NSMutableURLRequest requestWithURL:
url];
21     [request setValue:@"c925fbc1226c37b905a4d1e2a8cbb99"
forHTTPHeaderField:@"apikey"];
22     [NSURLConnection
sendAsynchronousRequest:request queue:[NSOperationQueue
mainQueue] completionHandler:^(NSURLResponse * _Nullable
response, NSData * _Nullable data, NSError * _Nullable
connectionError) {
23         NSString * dataStr = [[NSString alloc] initWithData:data

```

NSURLConnectionTest

```

2016-02-16 15:50:28.987 NSURLConnectionTest[2552:101949] 异步进行网络请求
2016-02-16 15:50:29.523 NSURLConnectionTest[2552:101949] {"showapi_res_code":
0,"showapi_res_error":"","showapi_res_body":{"allNum":17061,"allPages":854,"contentlist":
[{"ct":"2016-02-16 15:00:03.485","text":"那天同桌带女朋友一起上课，老师看了一眼说：大学里对象我不反对，但也得

```

图 4-10 异步进行网络请求的数据打印



提示

编程中的同步和异步可能与生活中的词汇意义有所偏差，读者只需记住：对于程序的执行方式来说，同步排队执行，异步同时执行。

4.1.4 使用 `NSURLConnection` 类通过代理回调的方式异步进行网络请求

在使用 `NSURLConnection` 进行网络请求时，`NSURLConnectionDataDelegate` 中约定了一些方法可以对其过程进行监听。首先在 `ViewController.m` 文件中添加遵守代理的相关代码，如下所示：

```
@interface ViewController ()<NSURLConnectionDataDelegate>
{
    NSMutableData * _data;
}
@end
```

上面代码声明了一个名为 `_data` 的可变数据对象，这个数据对象用来接收请求返回的数据。将 `viewDidLoad` 中的代码改写为如下所示：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSURL * url = [NSURL URLWithString:@"http://apis.baidu.com/showapi_open_bus/showapi_joke/joke_text?page=1"];
    NSMutableURLRequest * request = [NSMutableURLRequest requestWithURL:url];
    [request setValue:@"c925fbc1226c37b905a4d1e2a8cbbe99" forHTTPHeaderField:@"apikey"];
    NSURLConnection * connection = [[NSURLConnection alloc] initWithRequest:request delegate:self];
}
```

在 `ViewController.m` 中实现如下代理方法：

```
-(void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse *)response{
    NSLog(@"开始接收数据");
    //进行数据初始化
    _data = [[NSMutableData alloc] init];
}
-(void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data{
    [_data appendData:data];
}
-(void)connectionDidFinishLoading:(NSURLConnection *)connection{
```

```

    NSLog(@"接收数据完成");
    NSString * dataStr = [[NSString alloc] initWithData:_data encoding:NSUTF8StringEncoding];
    NSLog(@"%@", dataStr);
}

```

`connection:didReceiveResponse:`方法当接收到返回数据时会被调用,在其中进行了数据容器的初始化。

`connection:didReceiveData:`方法在接收数据过程中会多次被调用,其中会将接收到的数据传递进来。

`connectionDidFinishLoading:`方法在接收数据完成后被调用,在其中将接收到的完整数据转化为字符串进行了打印操作。

运行工程,打印调试区如图 4-11 所示,从打印内容可以看到, `NSURLConnection` 在接收返回数据的方法回调过程。

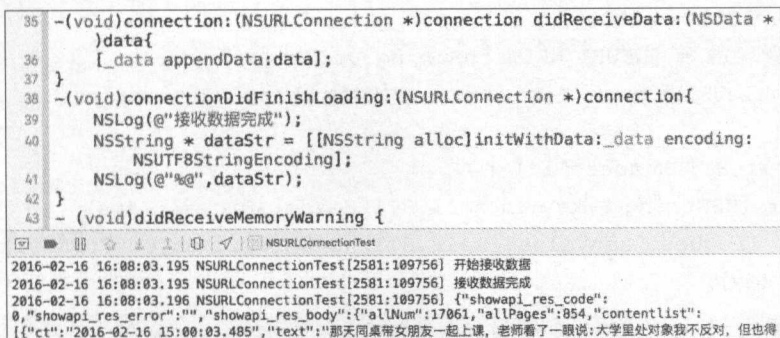


图 4-11 通过代理回调的方式进行网络请求的数据打印

4.2 设计封装一个更加易用的网络请求类

虽然系统的 `NSURLConnection` 设计的已经十分简洁,但是在面向应用时,多个请求依然需要重复编写 `NSURL`、`NSURLRequest` 的创建和请求头字段设置的相关代码。可以将其进一步进行封装使开发者在进行网络编程时更加方便。

使用 Xcode 创建一个名为 `MyRequestConnect` 的工程,在 `Info.plist` 文件中添加支持 HTTP 协议的字段。

4.2.1 设计自定义的网络请求连接类

网络请求连接类应该负责请求的发送和返回数据的传递。其应该作为一个工具将 `NSURL`、`NSURLRequest` 与 `NSURLConnection` 进行组合,为开发者在使用时简化一些中间步骤的代码。

使用 Xcode 创建一个新的文件,使其继承于 `NSObject`,命名为 `MyRequestConnection`。在 `MyRequestConnection.h` 中声明如下属性和方法:


```

@interface MyRequestConnection : NSObject
- (void)startRequestWithURLString:(NSString *)urlStr finished:(void (^)(BOOL success, NSData * data))finish;
@property(nonatomic, strong)NSDictionary* HTTPHeadersDic;
@property(nonatomic, strong)NSString * urlStr;
@end

```

其中，startRequestWithURLString:finished:方法用于开始请求任务，第 1 个参数设置请求的地址，第 2 个参数为获取到数据后带参数的 block，其中 success 参数决定请求是否成功，data 数据为请求的返回数据。HTTPHeadersDic 属性用于设置请求头文件参数。urlStr 为当前请求的地址。

在 MyRequestConnection.m 实现声明的方法，代码如下：

```

- (void)startRequestWithURLString:(NSString *)urlStr finished:(void (^)(BOOL, NSData *))finish{
    self.urlStr = urlStr;
    NSURL * url = [NSURL URLWithString:urlStr];
    NSMutableURLRequest * request = [NSMutableURLRequest requestWithURL:url];

    if (self.HTTPHeadersDic) {
        for (NSString * key in self.HTTPHeadersDic.allKeys) {
            [request setValue:[self.HTTPHeadersDic objectForKey:key] forHTTPHeaderField:key];
        }
    }

    [NSURLConnection sendAsynchronousRequest:request queue:[NSOperationQueue mainQueue] completionHandler:^(NSURLResponse * _Nullable response, NSData * _Nullable data, NSError * _Nullable connectionError) {
        if (connectionError==nil) {
            finish(YES, data);
        }else{
            NSLog(@"网络请求出错");
            finish(NO, data);
        }
    }];
}

```

上面的方法通过异步的方式进行网络请求，并将返回的数据通过 block 传递了出去。

4.2.2 设计自定义的网络请求连接管理类

设计好了请求连接类，还需要设计一个管理类对请求进行统一管理，这在开发中很有必要。一款完善的网络应用会发送很多请求，甚至同时发送很多请求，能够方便地对多请求进行管理是开发中必须考虑的问题。

创建一个新的类，使其继承于 NSObject，命名为 MyRequestManager。为了便于在程序中全局地管理网络请求，采用单例的设计模式来设计 MyRequestManager 类，首先在 MyRequestManager.h 中声明如下属性和方法：

```
@interface MyRequestManager : NSObject
+ (MyRequestManager *)sharedManager;
- (void)addRequestToManager:(NSString *)urlStr finished:(void (^)(BOOL success, NSData * data))finish;
@property(nonatomic, strong)NSDictionary * HTTPHeadersDic;
@end
```

类方法 sharedManager 用于获取单例对象，addRequestToManager:finish: 将一个网络请求添加进管理类的管理，HTTPHeadersDic 用于全局的设置请求头字段。

在 MyRequestManager.m 中引入 MyRequestConnection 类的头文件并声明一个可变数组用于存放请求的连接对象，如下所示：

```
#import "MyRequestConnection.h"
@implementation MyRequestManager
{
    NSMutableArray * _requestConnectionArray;
}
```

实现单例方法如下所示：

```
+ (MyRequestManager *)sharedManager{
    static MyRequestManager *manager = nil;
    static dispatch_once_t predicate;
    dispatch_once(&predicate, ^{
        manager = [[MyRequestManager alloc] init];
    });
    return manager;
}
```

使用 static 创建的静态变量可以保证不被释放，dispatch_once() 方法可以保证在程序执行过程中其内的代码只被执行 1 次，并且是线程安全的。

重写 init 初始化方法，在其中进行连接对象数组的初始化，如下所示：

```
- (instancetype)init
{
    self = [super init];
    if (self) {
        _requestConnectionArray = [[NSMutableArray alloc] init];
    }
    return self;
}
```

实现添加请求任务的方法如下所示：

```

- (void)addRequestToManager:(NSString *)urlStr finished:(void (^)(BOOL, NSData *))finish{
    for (MyRequestConnection * connection in _requestConnectionArray) {
        if ([connection.urlStr isEqualToString:urlStr]) {
            return;
        }
    }
    MyRequestConnection * connection = [[MyRequestConnection alloc] init];
    if (self.HTTPHeadersDic) {
        connection.HTTPHeadersDic = self.HTTPHeadersDic;
    }
    [_requestConnectionArray addObject:connection];
    [connection startRequestWithURLString:urlStr finished:^(BOOL success,
NSData *data) {
        [_requestConnectionArray removeObject:connection];
        finish(success, data);
    }];
}

```

在执行添加任务的方法时，先从任务数组中查询是否已经存在这个任务，如果已经存在，则不进行任何操作，这样可以避免相同的请求任务重复同时进行。接收到 `MyRequestConnection` 请求方法的返回数据后，将此请求和将获取的数据从管理数组中再次通过 block 传递出去。

通过上面两个类的设计，只是搭建了一个简单的自定义网络请求的框架，可以在 `MyRequestManager` 中继续添加取消某个请求任务和获取某个请求任务等方法，从请求管理数组中获取特定请求任务的方法可以通过为 `MyRequestConnection` 添加标识符属性来实现。网络请求框架的搭建涉及的内容十分繁杂，包括异常的处理、缓存的处理、数据加密解密的处理等，这些在第三方框架中都有完善的实现。读者在本章只需要了解网络请求框架的设计逻辑，在应用方面大多还是会采用第三方框架。

下面来看看在 `ViewController.m` 文件中 `MyRequestManager` 能否正常工作，首先引入头文件，代码如下：

```
#import "MyRequestManager.h"
```

在 `viewDidLoad` 方法中添加如下代码：

```

- (void)viewDidLoad {
    [super viewDidLoad];
    MyRequestManager * manager = [[MyRequestManager alloc] init];
    manager.HTTPHeadersDic = @{@"apikey":@"c925fbc1226c37b905a4d1e2a8cbbe99"};
}

```



```
[manager addRequestToManager:@"http://apis.baidu.com/showapi_open_bus/showapi_joke/joke_text?page=1" finished:^(BOOL success, NSData *data) {
    if (success) {
        NSString * dataStr = [[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding];
        NSLog(@"%@",dataStr);
    }
}];
```

运行工程，可以成功获取 API 服务提供的数据。经过封装的网络请求代码简单了许多，逻辑也更加清晰。并且在大多数情况下，一个应用程序中调用的所有网络请求都有相同的请求头字段，实际上在使用 MyRequestManager 进行请求头字段的设置时，只需要设置一次就可全局使用，十分方便。

4.3 JSON 类型数据的解析与数据模型的设计

API 服务返回的数据一般有两种格式类型，一种是 JSON 格式类型，一种是 XML 格式类型，相比之下 JSON 类型更加简洁方便，并已经成为移动 API 服务提供的主流数据类型。本节将介绍 iOS 开发中对 JSON 数据的解析以及将其映射为数据模型 Model 类的方法与过程。

4.3.1 JSON 数据简介

JSON 全称是 JavaScript Object Notation，是一种轻量级的数据交换格式，JSON 类型的数据书写方式类似字典，它是通过键值对的方法来组合数据，例如：

```
"name": "HuiShao"
```

JSON 值支持的数据类型及写法格式如下所示。

类型	写法格式
数字	直接书写
字符串	书写在双引号中
数组	书写在中括号中
字典	书写在大括号中
空对象	null

JSON 数据有两种结构，这两种结构取决于根数据类型是数组还是字典，如果 JSON 数据最外层用的是大括号，则此 JSON 数据为字典结构，在解析时应使用字典对象来接收，如果 JSON 数据最外层用的是中括号，则此 JSON 数据为数组结构，在解析时应使用数组对象来接收。

在 iOS 开发中要进行 JSON 解析，所解析的数据必须为严格的 JSON 格式，互联网上有许多工具和在线网站用来校验 JSON 数据，www.bejson.com 提供了免费的 JSON 数据校验服务，打开 www.bejson.com 网页，将如下数据粘贴进校验的数据区：

```
{
  "showapi_res_code": 0,
  "showapi_res_error": "",
  "showapi_res_body": {
    "allNum": 2980,
    "allPages": 149,
    "contentlist": [
      {
        "ct": "2015-08-13 13:10:26.149",
        "text": "新人发帖求过…… 媳妇最近怀孕了…天天这也不想吃那也不想吃…有一天发脾气要我给他做想吃的，结果做了好多还是没有想吃的…最后着急了大喊:再做不出我想吃的我就去大街上要饭……我想说:你吃什么自己都不知道我怎么做啊…唉…想想男人女人都不容易啊…",
        "title": "媳妇儿有了…",
        "type": 1
      },
      {
        "ct": "2015-08-13 13:10:26.149",
        "text": "为了让自己多活动，我把放在电脑桌上的零食拿到了外面的茶几上，这样最起码为了吃我也能走动走动。………现在我的零食经常会过期………",
        "title": "计划失败，吃货兼网虫的悲哀",
        "type": 1
      },
      {
        "ct": "2015-08-13 13:10:26.149",
        "text": "为了让自己多活动，我把放在电脑桌上的零食拿到了外面的茶几上，这样最起码为了吃我也能走动走动。………现在我的零食经常会过期………",
        "title": "计划失败，吃货兼网虫的悲哀",
        "type": 1
      }
    ]
  }
}
```

如图 4-12 所示，单击校验按钮，如果数据格式无误，校验结果区会出现 Valid JSON 的提示，如果格式有误，校验结果区会出现错误提示。

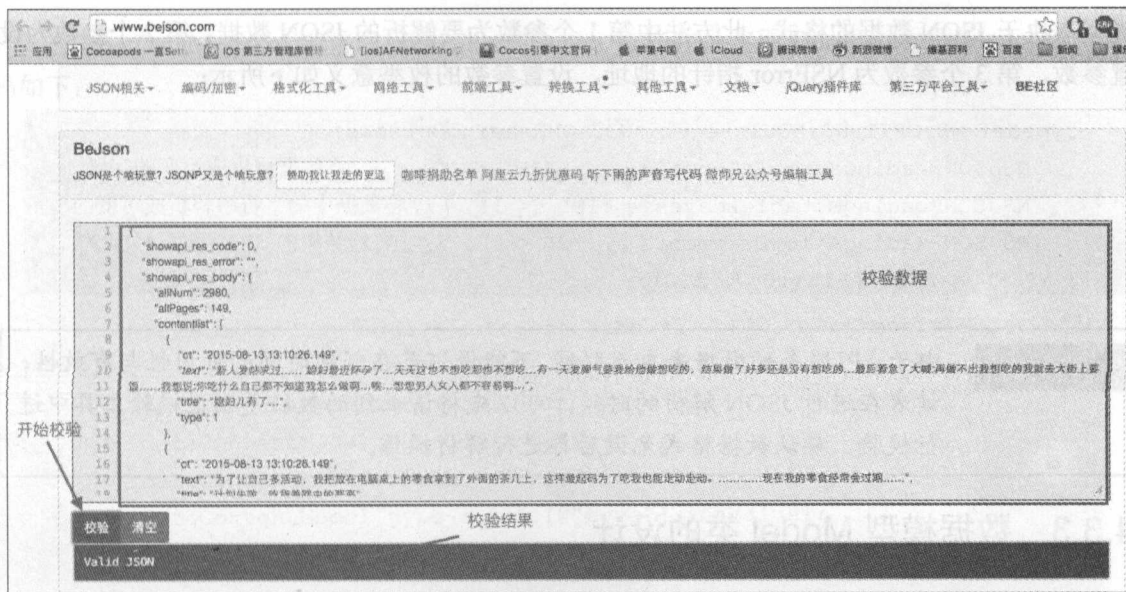


图 4-12 在线校验 JSON 数据

4.3.2 在 iOS 中解析 JSON 数据

在 iOS 中 JSON 数据的解析实际上就是将 JSON 格式的字符串映射为数组或者字典对象。在 iOS5 之后，Foundation 框架中提供 `NSJSONSerialization` 类来对 JSON 处理进行解析操作。

打开上节中创建的 `MyRequestConnect` 工程，将 `ViewController.m` 文件中的 `viewDidLoad` 方法改写为如下所示：

```

- (void)viewDidLoad {
    [super viewDidLoad];
    MyRequestManager * manager = [[MyRequestManager alloc] init];
    manager.HTTPHeadersDic = @{@"apikey":@"c925fbc1226c37b905a4d1e2a8cbbe99"};

    [manager addRequestToManager:@"http://apis.baidu.com/showapi_open_bus/showapi_joke/joke_text?page=2" finished:^(BOOL success, NSData *data) {
        if (success) {
            //进行数据解析
            NSError * error;
            NSDictionary * dataDic = [NSJSONSerialization JSONObjectWithData:data options:NSJSONReadingMutableContainers error:&error];
            NSLog(@"%@,%@",dataDic,error);
        }
    }];
}

```

在请求 API 服务的数据成功后，进行 JSON 数据的解析，`NSJSONSerialization.JSONObjectWithData:options:error:` 方法可以将 JSON 类型的 `NSData` 数据直接解析成数组或者字典对象，具体解析成的对象

类型取决于 JSON 数据的格式。此方法中第 1 个参数为要解析的 JSON 数据，第 2 个参数为设置参数，第 3 个参数为 NSError 指针的地址，设置参数的枚举意义如下所示：

```
typedef NS_OPTIONS(NSUInteger, NSJSONReadingOptions) {
    NSJSONReadingMutableContainers = (1UL << 0), //将数据解析成可变的容器
    NSJSONReadingMutableLeaves = (1UL << 1), //将数据中的字符串解析成可变的
    NSJSONReadingAllowFragments = (1UL << 2) //非容器结构的 JSON 数据解析
} NS_ENUM_AVAILABLE(10_7, 5_0);
```



提示

由于 API 服务的数据来自互联网，不能保证其在任何时间的可用性与有效性，读者在进行 JSON 解析的时候，可以先将请求到的数据复制进校验工具中进行校验，确认数据格式无误后再进行解析操作。

4.3.3 数据模型 Model 类的设计

对于 MVC 设计模式中的 3 个重要组成部分：M(Model)-V(View)-C(Controller)。View 层和 Controller 层在前面的章节都有介绍，本节将介绍 Model 层的基础：数据模型类的设计。

在 4.3.2 小节中，将从 API 服务获取到的数据解析成了一个容器数组或者字典，直接使用数组或者字典中的数据与 View 层进行绑定明显是不符合面向对象编程的要求的，因此需要将数组字典这类容器解析为具体的数据对象。

打开 MyRequestConnect 工程，在其中新建一个文件，使其继承于 NSObject，取名为 RootDataModel。分析 API 服务返回的数据格式，为了将需要的字段提取出来，在 RootDataModel.h 中声明如下对象：

```
@interface RootDataModel : NSObject
@property(nonatomic,assign)int allNum;
@property(nonatomic,assign)int allPages;
@property(nonatomic,assign)int currentPage;
@property(nonatomic,assign)int maxResult;
@property(nonatomic,strong)NSArray * contentlist;
@end
```

注意：数据模型中声明的对象名称要和接口返回数据中的字段名称一致。

通过分析数据可以发现，contentlist 数组中是每条数据具体的内容，可以将其再封装成一个数据模型，在 RootDataModel.h 中额外声明一个类，代码如下：

```
@interface DataContentistModel : NSObject
@property(nonatomic,strong)NSString * ct;
@property(nonatomic,strong)NSString * text;
@property(nonatomic,strong)NSString * title;
@property(nonatomic,assign)int type;
@end
```

在 RootDataModel.m 中重写 contentlist 的 setter 方法并且实现 DataContentistModel 类，代码如下：

```
@implementation RootDataModel
-(void)setContentlist:(NSArray *)contentlist{
    NSMutableArray * muArr = [[NSMutableArray alloc] init];
    for (int i=0; i<contentlist.count; i++) {
        NSDictionary * dic = contentlist[i];
        DataContentistModel * dataContentModel = [[DataContentistModel alloc] init];
        dataContentModel.ct = dic[@"ct"];
        dataContentModel.text = dic[@"text"];
        dataContentModel.title = dic[@"title"];
        dataContentModel.type = [dic[@"type"] intValue];
        [muArr addObject:dataContentModel];
    }
    _contentlist = [muArr copy];
}
@end

@implementation DataContentistModel
@end
```

在实现 RootDataModel 模型的 contentlist 属性的 setter 方法时，将字典数据转换为模型保存在数组。

在 ViewController.m 文件中进行接口数据与数据模型的转换，将 viewDidLoad 方法改写为如下所示：

```
-(void)viewDidLoad {
    [super viewDidLoad];
    MyRequestManager * manager = [[MyRequestManager alloc] init];
    manager.HTTPHeadersDic = @{@"apikey":@"c925fbc1226c37b905a4d1e2a8cbbe99"};
    [manager addRequestToManager:@"http://apis.baidu.com/showapi_open_bus/showapi_joke/joke_text?page=1" finished:^(BOOL success, NSData *data) {
        if (success) {
            //进行数据解析
            NSError * error;
            NSDictionary * dataDic = [NSJSONSerialization JSONObjectWithData:data options:NSJSONReadingMutableContainers error:&error];
            NSDictionary * modelDic = dataDic[@"showapi_res_body"];
            RootDataModel * dataModel = [[RootDataModel alloc] init];
            dataModel.allNum = [modelDic[@"allnum"] intValue];
            dataModel.allPages = [modelDic[@"allPages"] intValue];
        }
    }];
}
```

```

        dataModel.currentPage = [modelDic[@"currentPage"] intValue];
        dataModel.maxResult = [modelDic[@"maxResult"] intValue];
        dataModel.contentlist = modelDic[@"contentlist"];
        NSLog(@"%@", dataModel);
    }

    }
}

```

NSLog 方法实际只能打印出数据模型的地址，可以在 NSLog 处添加一个调试断点，运行工程，在 Xcode 调试区可以看到数据模型的结构，如图 4-13 所示。

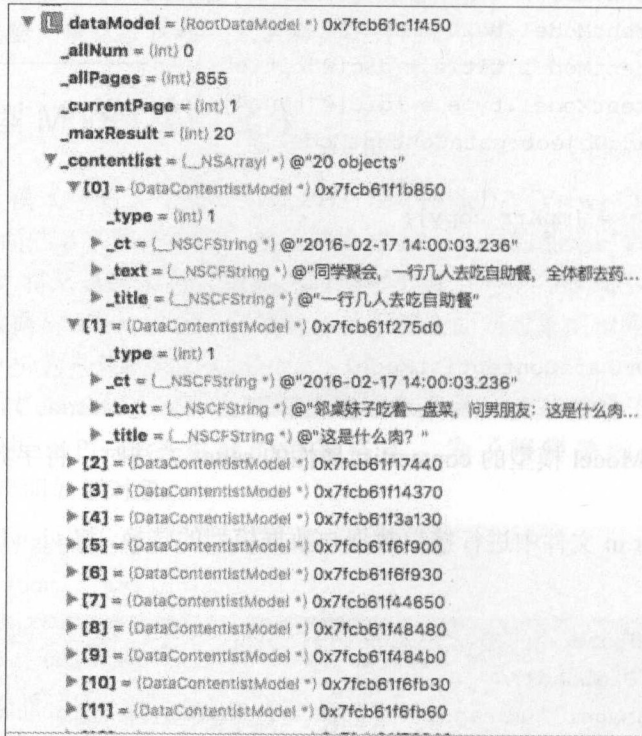


图 4-13 数据模型结构



提示

可以通过单击 Xcode 代码行标来添加断点，如图 4-14 所示。

```

27     NSDictionary * modelDic = dataDic[
28         RootDataModel * dataModel = [[RootD
29         dataModel.allNum = [modelDic[@"allN
30         dataModel.allPages = [modelDic[@"a
31         dataModel.currentPage = [modelDic[
32         dataModel.maxResult = [modelDic[@"
33         dataModel.contentlist = modelDic[
34         NSLog(@"%@", dataModel);
35     }
36
37     }
38 }
39

```

图 4-14 添加调试断点

在对数据模型中的属性进行赋值的时候,上面的代码采用分别调用 setter 方法的方式,这种方式需要编写许多机械化的代码,当数据模型十分复杂时,使用这种方式对数据模型进行初始化会耗费大量的时间和精力,iOS 中提供了一种更为简便的方式来对数据模型的属性进行复制操作,一般称其为 KVC (key-value-coding),即键值编码。

将 RootDataModel.m 文件中的代码修改为如下所示:

```
@implementation RootDataModel
-(void)setContentlist:(NSArray *)contentlist{
    NSMutableArray * muArr = [[NSMutableArray alloc] init];
    for (int i=0; i<contentlist.count; i++) {
        NSDictionary * dic = contentlist[i];
        DataContentistModel * dataContentModel = [[DataContentistModel alloc] init];
        //键值编码
        [dataContentModel setValuesForKeysWithDictionary:dic];
        [muArr addObject:dataContentModel];
    }
    _contentlist = [muArr copy];
}
-(void)setValue:(id)value forUndefinedKey:(NSString *)key{
}
@end

@implementation DataContentistModel
-(void)setValue:(id)value forUndefinedKey:(NSString *)key{
}
@end
```

setValuesForKeysWithDictionary:方法通过 KVC 的方式直接将字典映射成数据模型,需要注意的是,数据模型中的属性名称必须和字典中的键值一致。实现 setValue:forUndefinedKey:方法用于处理字典中的某个键值数据模型中并没有对应的属性这一情况。如果不重新实现这一方法,当字典中的键值和数据模型中的属性有不对应,程序会崩溃,实现了这一方法 KVC 在赋值时会自动跳过数据模型中没有定义属性的键值。将 ViewController.m 中的相关代码改写为如下所示:

```
-(void)viewDidLoad {
    [super viewDidLoad];
    MyRequestManager * manager = [[MyRequestManager alloc] init];
    manager.HTTPHeadersDic = @{@"apikey":@"c925fbc1226c37b905a4d1e2a8cbbe99"};
    [manager addRequestToManager:@"http://apis.baidu.com/showapi_open_bus/showapi_joke/joke_text?page=1" finished:^(BOOL success, NSData *data) {
        if (success) {
```

图 4-17 下载安装 CocoaPods 成功

```

        //进行数据解析
        NSError * error;
        NSDictionary * dataDic = [NSJSONSerialization JSONObjectWithData:
a:data options:NSJSONReadingMutableContainers error:&error];
        NSDictionary * modelDic = dataDic[@"showapi_res_body"];
        RootDataModel * dataModel = [[RootDataModel alloc] init];
        [dataModel setValuesForKeysWithDictionary:modelDic];
        NSLog(@"%@", dataModel);
    }
    }];
}

```

可以体会到使用 KVC 这种方法大大减少了开发者对数据模型赋值的工作量。

4.4 使用 CocoaPods 进行第三方库的管理

在实际的 iOS 开发中，借助第三方框架来实现部分功能是必不可少的。大多第三方框架都会将源码开放，使用第三方框架有如下几点优势：

- (1) 复杂的功能模块直接借助第三方框架实现，不必重复编写代码。
- (2) 开源的代码经过众多开发者的检验与完善，可靠性高。
- (3) 第三方框架往往对异常的处理和各个情况的考虑更加完整。

在项目中使用第三方框架时，也存在着许多不便，例如：

- (1) 互联网上第三方框架众多，官方地址难寻。
- (2) 第三方框架的编译环境可能与当前项目不一致，需要修改编译环境。
- (3) 某些第三方框架可能依赖系统库甚至其他第三方库，可能会引入麻烦。

CocoaPods 是一款非常优秀的管理第三方库的工具，使用 CocoaPods 可以方便地在项目中引入和管理第三方框架，CocoaPods 可以帮助开发者完成编译环境的设置、依赖库的下载和引入，除此之外，使用 CocoaPods 还可以方便地对第三方框架的版本进行管理，升级起来也十分方便。

4.4.1 在 MAC 上安装 CocoaPods

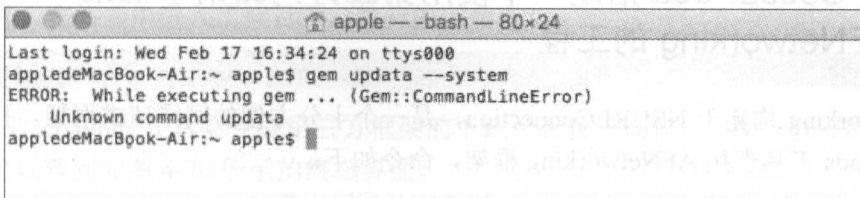
CocoaPods 虽然使用起来十分方便，但安装过程却有些繁琐，过程如下所示。

1. 升级 Ruby 环境

打开终端程序，在其中输入如下命令：

```
$gem update -system
```

如果没有权限，终端会显示如图 4-15 所示的界面。



```

apple — bash — 80x24
Last login: Wed Feb 17 16:34:24 on ttys000
appledeMacBook-Air:~ apple$ gem update --system
ERROR: While executing gem ... (Gem::CommandLineError)
Unknown command update
appledeMacBook-Air:~ apple$

```

图 4-15 没有权限时终端的提示界面

使用如下命名进行强制升级：

```
$sudo gem update --system
```

2. 下载安装 CocoaPods 工具

CocoaPods 工具的下载官网在国内访问速度十分慢，可以通过更换镜像的方法来处理。在终端输入以下两条命令：

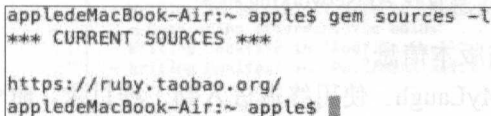
```
gem sources --removehttps://rubygems.org/
gem sources -ahttps://ruby.taobao.org/
```

在执行上面代码的时候，终端可能需要输入 MAC 密码，输入时终端不会有反应，读者不用担心，输入正确的密码敲击 return 键即可。

使用如下的命令检查镜像替换是否成功：

```
$ gem sources -l
```

如果镜像替换成功，终端中会出现如图 4-16 所示的界面。



```

appledeMacBook-Air:~ apple$ gem sources -l
*** CURRENT SOURCES ***
https://ruby.taobao.org/
appledeMacBook-Air:~ apple$

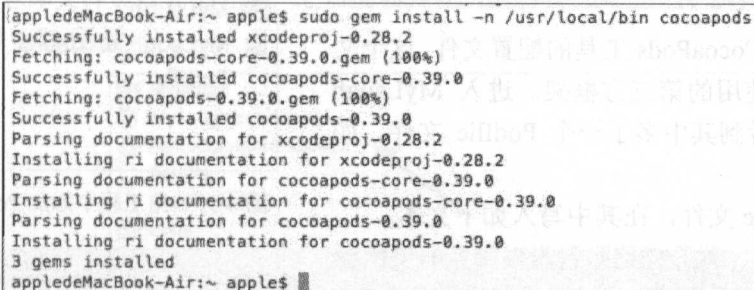
```

图 4-16 终端中替换 taobao 镜像成功的界面

在终端中输入如下命令，进行 CocoaPods 工具的下载和安装。

```
sudo gem install -n /usr/local/bin cocoapods
```

如果下载安装成功，终端界面如图 4-17 所示。



```

appledeMacBook-Air:~ apple$ sudo gem install -n /usr/local/bin cocoapods
Successfully installed xcodeproj-0.28.2
Fetching: cocoapods-core-0.39.0.gem (100%)
Successfully installed cocoapods-core-0.39.0
Fetching: cocoapods-0.39.0.gem (100%)
Successfully installed cocoapods-0.39.0
Parsing documentation for xcodeproj-0.28.2
Installing ri documentation for xcodeproj-0.28.2
Parsing documentation for cocoapods-core-0.39.0
Installing ri documentation for cocoapods-core-0.39.0
Parsing documentation for cocoapods-0.39.0
Installing ri documentation for cocoapods-0.39.0
3 gems installed
appledeMacBook-Air:~ apple$

```

图 4-17 下载安装 CocoaPods 成功

4.4.2 用 CocoaPods 搭建一个使用第三方网络请求框架 AFNetworking 的工程

AFNetworking 构建于 NSURLConnection，是一个十分方便的网络请求框架。打开终端，使用 CocoaPods 工具查找 AFNetworking 框架，命令如下：

```
$pod search afnetworking
```

会查找到许多与 AFNetworking 相关的第三方框架，终端界面如图 4-18 所示。

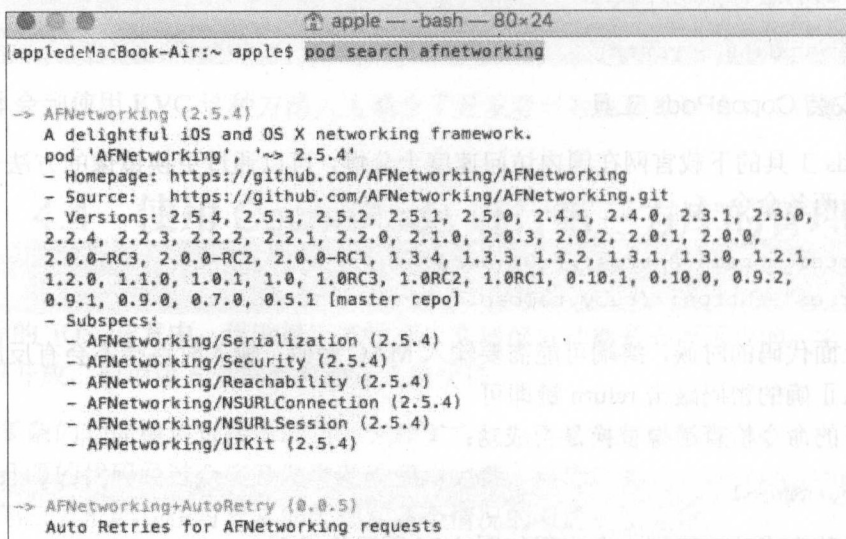


图 4-18 使用 CocoaPods 工具查找 AFNetworking 框架

从查找到的信息中可以获取框架的简介与版本信息。

使用 Xcode 创建一个新的工程，命名为 MyLaugh。使用终端进入到工程目录，命令如下：

```
$cd /Users/apple/Desktop/MyLaugh
```

上面命令中，cd 后面的路径为 MyLaugh 工程的路径，可以通过将 MyLaugh 工程文件夹拖进终端直接获取。

使用如下命令向工程中创建一个 Podfile 文件。

```
$ touch Podfile
```

Podfile 文件是 CocoaPods 工具的配置文件，这个文件中将配置工程要使用的第三方框架。进入 MyLaugh 工程文件夹，可以看到其中多了一个 Podfile 文件，如图 4-19 所示。

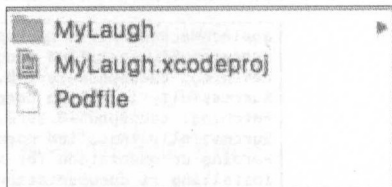


图 4-19 向工程中添加 Podfile 文件

双击打开 Podfile 文件，在其中写入如下文本。

```
platform:ios, '7.0'
pod 'AFNetworking', '~> 2.5.4'
```

第2行文本中的版本号为前面使用 CocoaPods 工具查询的版本号的最新版。保存并关闭 Podfile 文件，在终端中进入工程目录，执行如下命令。

```
$ pod install --verbose --no-repo-update
```

之后，CocoaPods 工具会完成第三方框架的下载和配置等操作，如果下载并配置第三方框架成功，可以看到如图 4-20 所示的终端界面。

```
> Git download
> Git download
$ /usr/bin/git clone https://github.com/AFNetworking/AFNetworking.git
/var/folders/pz/29dx9rvd6m77210g9jcqg5lw0000gn/T/d20160217-2545-1j0t3yo
--single-branch --depth 1 --branch 2.5.4 --recursive
Cloning into '/var/folders/pz/29dx9rvd6m77210g9jcqg5lw0000gn/T/d20160217-25
45-1j0t3yo'...
Note: checking out 'f38608bf2f7f5e13e673f3d77cf9b4fa5e4aad97'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

git checkout -b <new-branch-name>

> Copying AFNetworking from
'/Users/apple/Library/Caches/CocoaPods/Pods/Release/AFNetworking/2.5.4-05edc'
to 'Pods/AFNetworking'
- Running pre install hooks

Generating Pods project
- Creating Pods project
- Adding source files to Pods project
- Adding frameworks to Pods project
- Adding libraries to Pods project
- Adding resources to Pods project
- Linking headers
- Installing targets
- Installing target 'AFNetworking' iOS 7.0
- Installing target 'Pods' iOS 7.0
- Running post install hooks
- Writing Xcode project file to 'Pods/Pods.xcodeproj'
- Generating deterministic UUIDs
- Writing Lockfile in 'Podfile.lock'
- Writing Manifest in 'Pods/Manifest.lock'

Integrating client project

[!] Please close any current Xcode sessions and use 'MyLaugh.xcworkspace' for this
project from now on.

Integrating target 'Pods' ('MyLaugh.xcodeproj' project)
```

图 4-20 下载并配置成功第三方框架

打开 MyLaugh 工程文件夹，会发现其中多了一些文件和文件夹，CocoaPods 工具将第三方框架集成成为静态库文件引入到工程中，之后的项目开发都需要在工作控件 MyLaugh.xcworkspace 中进行，如图 4-21 所示。

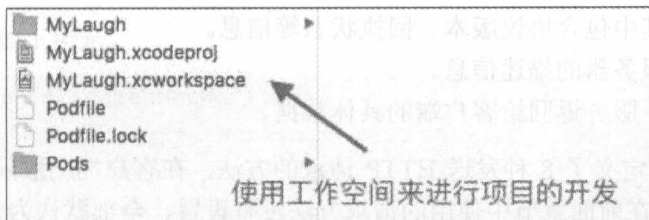


图 4-21 使用 CocoaPods 工具管理的工程结构

双击 MyLaugh.xcworkspace 文件，打开工作空间，Xcode 界面如图 4-22 所示，在 Xcode 的文件导航栏中有两个工程文件，一个为 MyLaugh，另一个为 Pods，MyLaugh 中的文件结构和以前没有区别，在其中进行代码的编写即可。

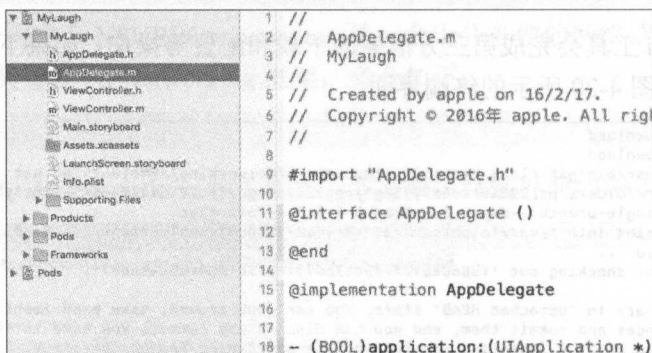


图 4-22 Xcode 工作空间中的目录结构

4.5 使用 AFNetworking 进行网络请求

使用 AFNetworking 框架进行网络请求要比使用 `NSURLConnection` 类方便许多也更加稳定和完美。

4.5.1 详解 HTTP/HTTPS 协议

前面章节已经介绍了在 iOS 中进行网络请求的一些方法，但并没有对 HTTP/HTTPS 协议做过多的介绍，能够将 iOS 网络编程技术熟练地应用于开发，则进一步地了解 HTTP/HTTPS 协议是必经之路。

一个完整的 HTTP 请求和响应中应包含 3 部分内容，在发送 HTTP 请求时应包含的 3 部分内容为：

- (1) 请求行。其中包含请求方法、请求路径、协议版本等信息。
- (2) 请求头。客户端的地址和环境描述、身份验证信息。
- (3) 请求体。客户端发送的具体数据。

在回执的响应中包含的 3 部分内容为：

- (1) 状态行。其中包含协议版本、回执状态等信息。
- (2) 响应头。服务器的描述信息。
- (3) 返回内容。服务返回给客户端的具体数据。

在 HTTP 协议中定义了 8 种发送 HTTP 请求的方法，在客户端一般只会用到两种：GET 请求和 POST 请求。在前面章节中使用的请求方法没有设置、全部默认为 GET 请求，具体采用哪种请求方法取决于服务端，客户端只需根据服务端的要求发送即可。

GET 请求参数直接放在请求地址中, 参数和地址使用 “?” 符号隔开, 参数与参数之间使用 “&” 符号隔开, 例如在前面章节请求数据时, 使用 GET 的请求的路径如下:

```
http://apis.baidu.com/showapi_open_bus/showapi_joke/joke_text?page=1
```

其中有一个参数 `page` 等于 1, 代表请求所有数据中的第 1 页。

POST 请求参数不放在请求路径中, 而是放在请求体中直接进行传输。

虽然对客户端来说, GET 请求和 POST 请求都可以进行 API 接口数据的获取, 但还有一些差异:

- (1) GET 请求无法传递大量的数据。
- (2) GET 请求的参数直接放在地址 URL 中, 安全性差。
- (3) GET 请求一般只用于获取数据。
- (4) POST 请求多用于修改, 增加, 删除数据。

4.5.2 使用 AFNetworking 进行网络请求

使用 Xcode 打开 MyLaughe 工作空间, 在 MyLaugh 项目的 `info.plist` 文件中添加支持 HTTP 请求的键值对。创建一个新的文件, 使其继承于 `NSObject`, 命名为 `MyNetworking`。

在 `MyNetworking.h` 文件中声明一些属性和方法, 代码如下:

```
@property(nonatomic, strong) NSDictionary * HTTPHeadersDic;
+ (MyNetWorking *)sharedNetWorking;
+ (void)getRequestWithURLString:(NSString *)urlStr finish:(void (^)(BOOL success, NSData *data))finish;
+ (void)postRequestWithURLString:(NSString *)urlStr paramDic:(NSDictionary *)param finish:(void (^)(BOOL success, NSData *data))finish;
@end
```

`MyNetworking` 采用单例的设计模式, 类方法 `sharedNetWorking` 用于获取单例对象。`getRequestWithURLString:finish:` 方法用于发起 GET 请求。`postRequestWithURLString:paramDic:finish:` 方法用于发起 POST 请求, 本方法中的 `param` 参数用于设置请求体中的参数。

在 `MyNetworking.m` 文件中引入 `AFNetworking` 的头文件, 代码如下:

```
#import <AFNetworking/AFNetworking.h>
```

由于 `CocoaPods` 是采用静态库的方式管理第三方框架的, 这里在引入头文件时使用的是 `<>` 而不是 `""`。

实现单例方法如下所示。

```
+ (MyNetWorking *)sharedNetWorking{
    static MyNetWorking *netWorking = nil;
    static dispatch_once_t predicate;
    dispatch_once(&predicate, ^{
        netWorking = [[MyNetWorking alloc] init];
    });
}
```

```

    });
    return netWorking;
}

```

实现发送 GET 请求与 POST 请求的方法如下所示。

```

+ (void)getRequestWithURLString:(NSString *)urlStr finish:(void (^)(BOOL,
NSData *))finish{
    urlStr = [urlStr stringByAddingPercentEscapesUsingEncoding:NSUTF8Stri
ngEncoding];
    AFHTTPRequestOperationManager * manager = [AFHTTPRequestOperationManag
er manager];
    manager.responseSerializer = [AFHTTPResponseSerializer serializer];
    if ([MyNetWorking sharedNetWorking].HTTPHeadersDic) {
        for (NSString * key in [MyNetWorking sharedNetWorking].HTTPHeadersDic) {
            [manager.requestSerializer setValue:[MyNetWorking sharedNetWork
ing].HTTPHeadersDic[key] forHTTPHeaderField:key];
        }
    }
    [manager GET:urlStr parameters:nil success:^(AFHTTPRequestOperation *o
peration, id responseObject) {
        finish(YES, responseObject);
    } failure:^(AFHTTPRequestOperation *operation, NSError *error) {
        finish(NO, nil);
    }];
}

+ (void)postRequestWithURLString:(NSString *)urlStr paramDic:(NSDictionary
*)param finish:(void (^)(BOOL, NSData *))finish{
    urlStr = [urlStr stringByAddingPercentEscapesUsingEncoding:NSUTF8Stri
ngEncoding];
    AFHTTPRequestOperationManager * manager = [AFHTTPRequestOperationManag
er manager];
    manager.responseSerializer = [AFHTTPResponseSerializer serializer];
    if ([MyNetWorking sharedNetWorking].HTTPHeadersDic) {
        for (NSString * key in [MyNetWorking sharedNetWorking].HTTPHeadersD
ic) {
            [manager.requestSerializer setValue:[MyNetWorking sharedNetWork
ing].HTTPHeadersDic[key] forHTTPHeaderField:key];
        }
    }
    [manager POST:urlStr parameters:param success:^(AFHTTPRequestOperation
n *operation, id responseObject) {
        finish(YES, responseObject);
    } failure:^(AFHTTPRequestOperation *operation, NSError *error) {

```

```

        finish(NO, nil);
    }];
}

```

上面的方法中，调用字符串的 `stringByAddingPercentEscapesUsingEncoding:` 方法用于处理请求地址中的中文字符。`AFHTTPRequestOperationManager` 类是 `AFNetworking` 框架中用于管理 HTTP 请求的类，下面的代码用于初始化类对象。

```

AFHTTPRequestOperationManager * manager = [AFHTTPRequestOperationManager
manager];

```

下面代码用于设置返回的数据为原始数据，`AFNetworking` 不做额外处理。

```

manager.responseSerializer = [AFHTTPResponseSerializer serializer];

```

下面代码用于设置请求头文件参数。

```

[manager.requestSerializer setValue:[MyNetWorking sharedNetWorking].HTTPH
eadersDic[key] forKey:@"HTTPHeaderField:key"];

```

为验证 `MyNetworking` 工作是否正常，在 `ViewController.m` 文件的 `viewDidLoad` 方法中添加如下代码：

```

- (void)viewDidLoad {
    [super viewDidLoad];
    [MyNetWorking sharedNetWorking].HTTPHeadersDic = @{@"apikey":@"c925fb
c1226c37b905a4d1e2a8cbb99"};
    [MyNetWorking getRequestWithURLString:@"http://apis.baidu.com/showapi
_open_bus/showapi_joke/joke_text?page=1" finish:^(BOOL success, NSData *data)
    {
        if (success) {
            NSDictionary * dic = [NSJSONSerialization JSONObjectWithData:da
ta options:NSJSONReadingMutableContainers error:nil];
            NSLog(@"%@", dic);
        }
    }];
}

```

运行工程，可以看到打印出请求到的 API 服务数据。

4.6 实战：开发“笑一笑”应用程序

本节将使用百度的免费 API 服务接口开发一款休闲娱乐应用“笑一笑”。“笑一笑”应用主要功能分为两部分，一部分是提供实时更新的文本笑话，一部分是提供实时的幽默图片。

4.6.1 工程项目框架的搭建

打开前面章节创建的 MyLaugh 工作空间。将 MyLaugh 项目的 UI 框架改为以 UITabBarController 为入口，单击 Main.storyboard 文件，删掉其中自带的 View Controller，向界面中拖入一个 TabBarController，如图 4-23 所示。

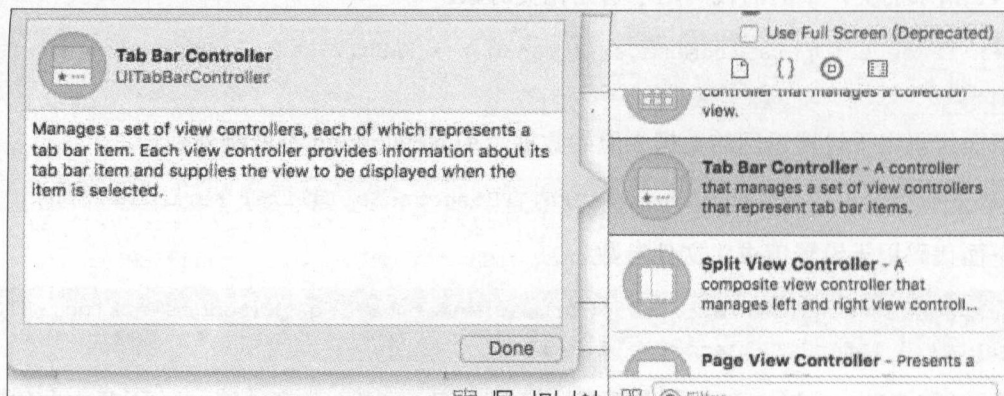


图 4-23 向 storyboard 中拖入一个 TabBarController

删掉 TabBarController 自带的两个子视图控制器，向界面中拖入两个导航控制器作为 TabBarController 的两个子控制器。分别设置 TabBarController 中两个子视图控制器的 TabBarItem，如图 4-24 与图 4-25 所示。

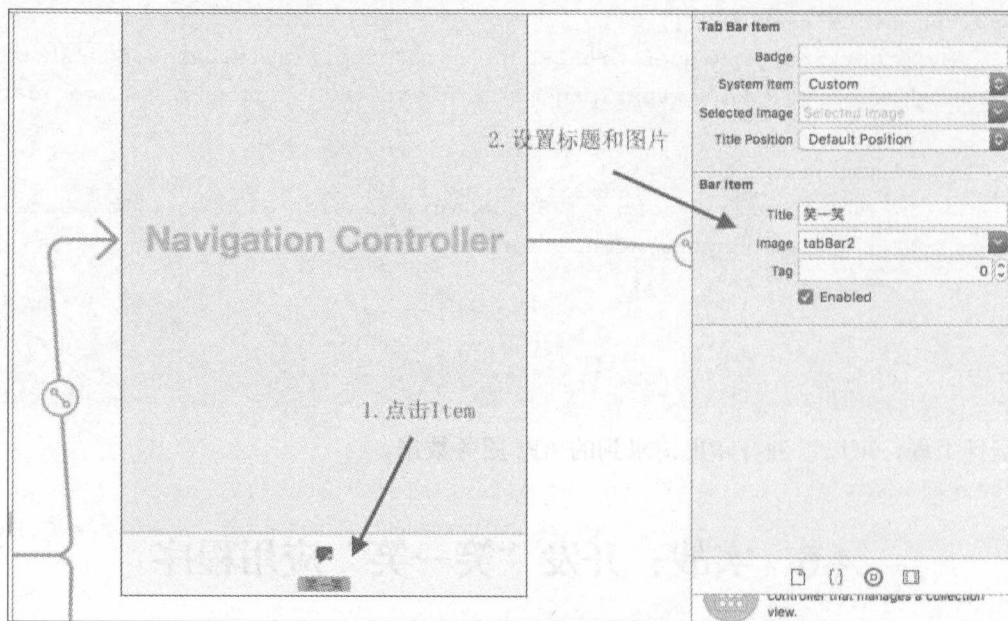


图 4-24 设置 TabBarItem

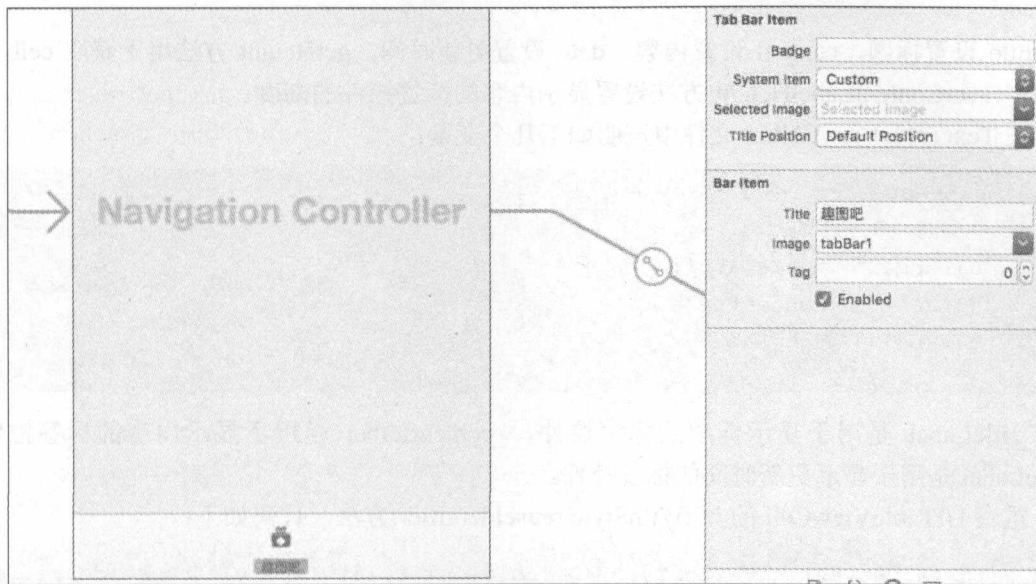


图 4-25 设置 TabBarItem

将 ViewController 类的继承关系修改为继承于 UITableViewController, ViewController.h 文件修改为如下所示:

```
@interface ViewController : UITableViewController
@end
```

将 TabBarController 中“笑一笑”Item 对应的导航的根视图控制器与其关联,如图 4-26 所示。

创建一个新的文件,取名为 ImageTableViewController,使其继承于 UITableViewController,作为幽默图片界面的视图控制器,并将 TabBarController 的“趣图吧”Item 对应导航的根视图控制器与之关联。

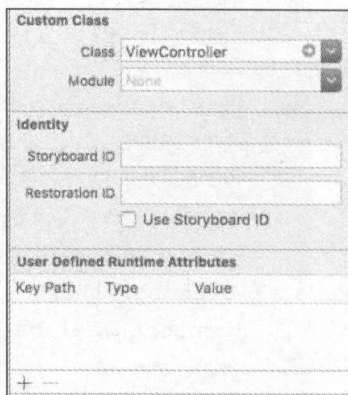


图 4-26 进行视图控制器的关联

4.6.2 “笑一笑”界面数据载体 cell 的设计

“笑一笑”界面主要展示文本笑话,对于 API 服务中提供的数据,有 3 个字段需要界面来展示,分别为标题、内容和更新时间。使用 Xcode 创建一个命名为 TextTableViewCell 的类,使其继承自 UITableViewCell,在其头文件中声明如下属性和方法:

```
@interface TextTableViewCell : UITableViewCell
@property(nonatomic,strong)NSString * title;
@property(nonatomic,strong)NSString * content;
@property(nonatomic,strong)NSString * date;
-(CGFloat)getHeight;
-(void)setContentLabelHeight:(CGFloat)height;
@end
```

title 设置标题, content 设置内容, date 设置更新时间, getHeight 方法用于获取 cell 自身的高度, setContentLabelHeight:方法设置显示内容的标签控件的高度。

在 TextTableViewCell.m 文件中声明如下几个变量:

```
@implementation TextTableViewCell
{
    UILabel * _titleLabel;
    UILabel * _contentLabel;
    UILabel * _dateLabel;
}
```

_titleLabel 是用于显示标题的标签控件, _contentLabel 是用于显示内容的标签控件, _dateLabel 是用于显示更新时间的标签控件。

重写 UITableViewCell 的 initWithStyle:reuseIdentifier:方法, 代码如下:

```
- (instancetype)initWithStyle:(UITableViewCellStyle)style reuseIdentifier:
(nullable NSString *)reuseIdentifier{
    self = [super initWithStyle:style reuseIdentifier:reuseIdentifier];
    _titleLabel = [[UILabel alloc] initWithFrame:CGRectMake(0, 5, self.frame.size.width, 21)];
    _titleLabel.backgroundColor = [UIColor clearColor];
    _titleLabel.textAlignment = NSTextAlignmentCenter;
    _titleLabel.textColor = [UIColor purpleColor];
    _titleLabel.font = [UIFont italicSystemFontOfSize:17];
    _contentLabel = [[UILabel alloc] initWithFrame:CGRectMake(0, 30, self.frame.size.width, 21)];
    _contentLabel.backgroundColor = [UIColor clearColor];
    _contentLabel.numberOfLines=0;
    _dateLabel = [[UILabel alloc] initWithFrame:CGRectMake(0, 30, self.frame.size.width, 21)];
    _dateLabel.textAlignment = NSTextAlignmentRight;
    [self.contentView addSubview:_titleLabel];
    [self.contentView addSubview:_contentLabel];
    [self.contentView addSubview:_dateLabel];
    self.selectionStyle = UITableViewCellSelectionStyleNone;
    return self;
}
```

上面代码中对 cell 中的几个标签控件做了初始化处理。需要注意的是, 在 cell 上添加子视图时, 要添加在 cell 的 contentView 上。

在 TextTableViewCell.m 中再实现如下方法:

```
-(void)setTitle:(NSString *)title{
    _title= title;
}
```



```

    _titleLabel.text = title;
}
-(void)setContent:(NSString *)content{
    _content = content;
    _contentLabel.text= content;

}
-(void)setDate:(NSString *)date{
    _date= date;
    _dateLabel.text = date;
}
-(void)setContentLabelHeight:(CGFloat)height{
    _contentLabel.frame=CGRectMake(0, 30, _contentLabel.frame.size.width,
height);
    _dateLabel.frame = CGRectMake(0, 35+height, _dateLabel.frame.size.widt
h, 21);
}
-(CGFloat)getHeight{
    //计算文字高度
    NSDictionary *attribute = @{NSFontAttributeName: _contentLabel.font};
    CGSize retSize = [_contentLabel.text boundingRectWithSize:CGSizeMake(s
elf.frame.size.width, MAXFLOAT)
                        options:
                            NSStringDrawingTruncatesLastVisibleLine |
                            NSStringDrawingUsesLineFragmentOrigin |
                            NSStringDrawingUsesFontLeading
                        attributes:attribute
                        context:nil].size;
    _contentLabel.frame=CGRectMake(0, 30, _contentLabel.frame.size.width,
retSize.height);
    _dateLabel.frame=CGRectMake(0, 35+_contentLabel.frame.size.height, _d
ateLabel.frame.size.width, 21);
    return 30+_contentLabel.frame.size.height+10+21+5;
}

```

setTitle:方法、setContent:方法和 setDate:方法都是对相应标签控件进行了赋值操作，setContentLabelHeight:方法重设内容标签_contentLabel 的尺寸。getHeight 方法对 cell 本身的高度进行计算，其中 boundingRectWithSize:options:attribute:context:方法用于计算文字的高度。

4.6.3 “笑一笑”界面的搭建

在 ViewController.m 文件中引入一些头文件，如下所示：

```
#import "MyNetWorking.h"
#import "RootDataModel.h"
#import "TextTableViewCell.h"
```

RootDataModel 类是 API 接口返回数据映射成的数据模型，这个数据模型中除了 API 接口数据中的一些字段外，需要再添加一些与 UI 相关的字段，便于开发者在搭建 UI 界面时使用，RootDataModel.h 文件的内容如下：

```
#import <UIKit/UIKit.h>
@interface RootDataModel : NSObject
@property(nonatomic,assign)int allNum;
@property(nonatomic,assign)int allPages;
@property(nonatomic,assign)int currentPage;
@property(nonatomic,assign)int maxResult;
@property(nonatomic,strong)NSArray * contentlist;
@end
@interface DataContentistModel : NSObject
@property(nonatomic,strong)NSString * ct;
@property(nonatomic,strong)NSString * text;
@property(nonatomic,strong)NSString * title;
@property(nonatomic,assign)int type;
@property(nonatomic,strong)NSString * img;
@property(nonatomic,assign)CGFloat height;
@end
```

DataContentistModel 是内容数据模型，其中除了 API 返回数据中的字段外，还额外添加了一个 height 字段用于存储此数据模型载体 cell 的高度。RootDataModel.m 中的内容如下：

```
@implementation RootDataModel
-(void)setContentlist:(NSArray *)contentlist{
    NSMutableArray * muArr = [[NSMutableArray alloc] init];
    for (int i=0; i<contentlist.count; i++) {
        NSDictionary * dic = contentlist[i];
        DataContentistModel * dataContentModel = [[DataContentistModel alloc] init];
        //键值编码
        [dataContentModel setValuesForKeysWithDictionary:dic];
        [muArr addObject:dataContentModel];
    }
    _contentlist = [muArr copy];
}
-(void)setValue:(id)value forUndefinedKey:(NSString *)key{
}
@end
```

```

@implementation DataContentistModel
-(void)setValue:(id) value forUndefinedKey:(NSString *)key{
}
@end

```

在 ViewController.m 中在定义一个宏来代理接口地址，便于在开发中请求的书写。

```

#define TEXTURL @"http://apis.baidu.com/showapi_open_bus/showapi_joke/joke_text?page=%d"

```



提示

宏定义是一种简单的代码替换，如上所示，在代码中使用 TEXTURL 就代表使用后面的 url 地址字符串。

在 ViewController.m 中声明一些变量，如下所示。

```

@interface ViewController ()
{
    int _currentPage;
    NSMutableArray * _dataArray;
    UIButton * _moreBtn;
}
@end

```

由于接口数据有分页，_currentPage 字段表示当前加载的数据页数，_dataArray 是数据源数组，_moreBtn 为加载更多按钮。

在 ViewController.m 文件的 viewDidLoad 方法中编写如下代码：

```

- (void)viewDidLoad {
    [super viewDidLoad];
    _currentPage = 1;
    self.title = @"笑一笑";
    [MyNetWorking sharedNetWorking].HTTPHeadersDic = @{@"apikey":@"c925fb
c1226c37b905a4d1e2a8cbb99"};
    _dataArray = [[NSMutableArray alloc] init];
    _moreBtn = [UIButton buttonWithType:UIButtonTypeSystem];
    _moreBtn.frame=CGRectMake(0, 0, self.view.frame.size.width,30);
    [_moreBtn setTitle:@"加载更多~~" forState:UIControlStateNormal];
    [_moreBtn addTarget:self action:@selector(readMore) forControlEvents:
    UIControlEventTouchUpInside];
    self.tableView.tableFooterView = _moreBtn;
    [self getDataWithPage:_currentPage];
    [self creatRefresh];
}

```


上面代码中对请求设置、界面标题、加载页数和加载更多按钮进行了初始化的操作并将加载更多按钮设置成了表格视图的尾视图，当表格滑动到底端时，这个按钮就会被显现出来。`getDataWithPage:`方法将从 API 接口获取数据，`creatRefresh` 方法对下拉刷新操作进行初始化，关于下拉刷新会在下一节进行介绍，读者在此可以先将 `creatRefresh` 方法进行空实现，不会妨碍正常代码的运行。

首先实现 `UITableViewDelegate` 和 `UITableViewDataSource` 的相关方法，代码如下：

```

- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath{
    DataContentistModel * model = _dataArray[indexPath.row];
    return model.height;
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section{
    return _dataArray.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath{
    TextTableViewCell * cell = [tableView dequeueReusableCellWithIdentifier:@"TextTableViewCell"];
    if (cell==nil) {
        cell=[[TextTableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"TextTableViewCell"];
    }
    DataContentistModel * model = _dataArray[indexPath.row];
    cell.title = model.title;
    cell.content = model.text;
    cell.date = [NSString stringWithFormat:@"更新时间:%@",[model.ct substringToIndex:10]];
    if (model.height==0) {
        model.height = [cell getHeight];
    }else{
        [cell setContentLabelHeight:model.height-56];
    }
    return cell;
}

```

`tableView:heightForRowAtIndexPath:`方法中数据源数组中获取对应的数据模型，从数据模型中取 `cell` 的高度进行返回。

tableView:cellForRowAtIndexPath:方法通过数据源的数据对 cell 进行了设置,在设置完 cell 之后对数据模型中 height 字段进行判断,如果此数据模型没有记录 cell 的高度,则通过 TextTableViewCell 的 getHeight 方法将计算出的高度赋值给数据模型,如果数据模型中已经有 cell 的高度,则使用 TextTableViewCell 的 setContentLabelHeight 方法对 cell 中的内容标签控件的高度进行设置。

实现 getDataWithPage:方法,代码如下所示:

```
-(void)getDataWithPage:(int)page{
    [MyNetWorking getRequestWithURLString:[NSString stringWithFormat:TEXT
URL,page] finish:^(BOOL success, NSData *data) {
        if (success) {
            _currentPage++;
            NSDictionary * dataDic = [NSJSONSerialization JSONObjectWithData:
data options:NSJSONReadingMutableContainers error:nil];
            if (dataDic==nil) {
                //如果因为数据的问题 不能解析 跳过此页数据
                [self getDataWithPage:_currentPage];
            }else{
                NSDictionary * dic = dataDic[@"showapi_res_body"];
                RootDataModel * model = [[RootDataModel alloc] init];
                [model setValuesForKeysWithDictionary:dic];
                [_dataArray addObjectsFromArray:model.contentlist];
                [self.tableView reloadData];
            }
        }else{
            //进行网络问题的提示
            UIAlertController * alert = [UIAlertController alertControllerW
ithTitle:@"温馨提示" message:@"网络或服务异常" preferredStyle:UIAlertControlle
rStyleAlert];
            UIAlertAction * action = [UIAlertAction actionWithTitle:@"OK" s
tyle:UIAlertActionStyleCancel handler:nil];
            [alert addAction:action];
            [self presentViewController:alert animated:YES completion:nil];
        }
    }];
}
```

getDataWithPage:方法使用 MyNetWorking 进行 API 服务数据的请求,当请求数据成功后,将 _currentPage 的值加 1 并且把返回数据进行解析存入数据源数组。如果请求失败,则进行弹框提示用户网络错误。

运行工程,可以看到如图 4-27 所示的界面效果,此时“笑一笑”界面的基本功能都已经实现,开发者还需实现下拉刷新和加载更多功能。



图 4-27 “笑一笑”界面效果

4.6.4 实现下拉刷新与加载更多功能

下拉刷新操作是许多应用都会采用的一种用户触发的刷新操作方式。第三方刷新框架 MJRefresh 是一个十分强大易用的下拉刷新库，线上的众多商业应用都是采用这个第三方库来实现的刷新操作。实际上，在 iOS6 之后，原生的 iOS 框架也提供了下拉刷新的 UI 效果，在 UITableViewController 中就可以使用。

单击 Main.storyboard 文件，将两个导航控制器的根视图控制器的下拉刷新属性设置为允许，如图 4-28 所示。

在 ViewController.m 文件中实现 creatRefresh 方法，代码如下所示：

```
-(void)creatRefresh{
    self.refreshControl.tintColor = [UIColor greenColor];
    [self.refreshControl addTarget:self action:@selector(change:) forControlEvents:UIControlEventValueChanged];
}
```

refreshControl 是 UITableViewController 自带的一个刷新控件，其类似小风火轮，当用户将界面向下拉到一定尺度时会触发旋转。refreshControl 属性继承自 UIControl，可以为其添加监听事件，当控件被激活为刷新状态后会调用 change: 方法。

change: 方法的实现如下所示：

```
-(void)change:(UIRefreshControl *)refreshControl{
    _moreBtn.enabled=NO;
    [MyNetWorking getRequestWithURLString:[NSString stringWithFormat:TEXT
URL,1] finish:^(BOOL success, NSData *data) {
        if (success) {
            NSDictionary * dataDic = [NSJSONSerialization JSONObjectWithData:
data options:NSJSONReadingMutableContainers error:nil];
        }
    }];
}
```



```

if (dataDic==nil) {
    //如果因为数据的问题 不能解析 跳过此页数据
}else{
    NSDictionary * dic = dataDic[@"showapi_res_body"];
    RootDataModel * model = [[RootDataModel alloc] init];
    [model setValuesForKeysWithDictionary:dic];
    [_dataArray replaceObjectsInRange:NSMakeRange(0, model.contentlist.count) withObjectsFromArray:model.contentlist];
    [self.tableView reloadData];
}
}else{
    //进行网络问题的提示
    UIAlertController * alert = [UIAlertController alertControllerWithTitle:@"温馨提示" message:@"网络或服务异常" preferredStyle:UIAlertControllerStyleAlert];

    UIAlertAction * action = [UIAlertAction actionWithTitle:@"OK" style:UIAlertActionStyleCancel handler:nil];
    [alert addAction:action];
    [self presentViewController:alert animated:YES completion:nil];
}
_moreBtn.enabled=YES;
[self.refreshControl endRefreshing];
}];
}

```

刷新数据的方法与获取数据的方法类似，只是刷新数据始终请求第 1 页的数据，并用请求到的数据将数据源中的数据做替换。

运行工程，向下拉动表格视图，刷新效果如图 4-29 所示。

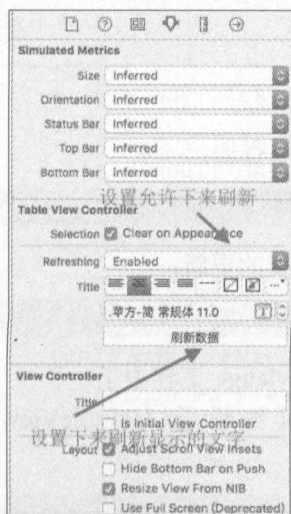


图 4-28 设置 UITableViewController 允许下拉刷新

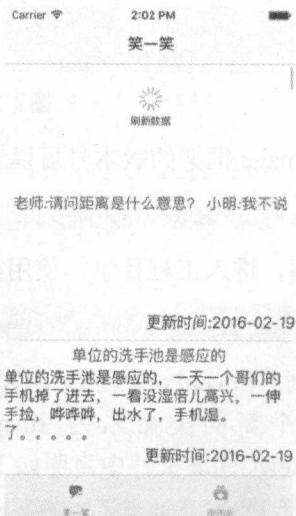


图 4-29 为 UITableView 添加下拉刷新操作

对于加载更多，readMore 方法的实现如下所示。

```
-(void)readMore{
    [self getDataWithPage:_currentPage];
}
```

因为每次请求数据成功后 _currentPage 参数都会加 1，readMore 方法直接加载当前 _currentPage 页的数据即可。

4.6.5 “趣图吧”界面数据载体 cell 的设计

和文本笑话不同时，幽默图片从 API 获取到的数据是一个图片的 URL 地址，开发者需要根据这个 URL 地址再获取相应的图片数据。由于下载图片一般会有很大的数据量，使用同步的加载方式难免会使界面卡死，对于加载网络图片需求的处理，一般会采用 SDWebImage 这个第三方框架来进行开发。

打开工程文件下的 Podfile 文件，向其中添加如下命令：

```
pod 'SDWebImage', '~> 3.7.3'
```

CocoaPods 文件的配置如图 4-30 所示。

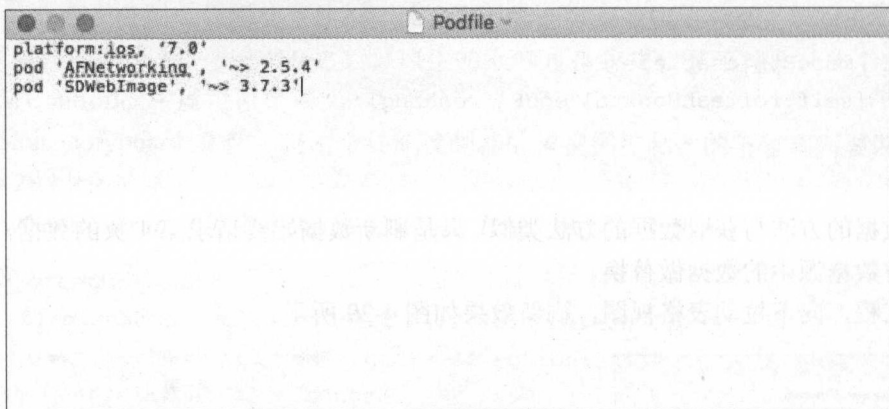


图 4-30 CocoaPods 文件的配置

SDWebImage 框架的版本号可以通过在终端使用如下命令查询：

```
$ pod search SDWebImage
```

打开终端，进入工程目录，使用如下命令进行第三方框架的更新：

```
$ pod install --verbose --no-repo-update
```

等待片刻，CocoaPods 就将 SDWebImage 框架添加进 Mylaugh 工程中了。

创建一个新的类命名为 ImageTableViewCell，使其继承于 UITableViewCell。在 ImageTableViewCell.h 文件中声明如下属性和方法：

```
@interface ImageTableViewCell : UITableViewCell
@property(nonatomic,strong)NSString * title;
```

```

@property(n nonatomic, strong) NSString * content;
@property(n nonatomic, strong) NSString * date;
@property(n nonatomic, assign) int indexRow;
- (CGFloat) getHeight;
@end

```

在 ImageTableViewCell.m 中引入异步加载网络图片的头文件，如下所示：

```
#import <UIImageView+WebCache.h>
```

在 ImageTableViewCell.m 中声明如下变量：

```

@implementation ImageTableViewCell
{
    UIImageView * _webImageView;
    UILabel * _titleLabel;
    UILabel * _dateLabel;
    CGFloat _cellHeight;
}

```

_webImageView 用于显示内容图片，_cellHeight 为 cell 本身的高度。

重写 initWithStyle:reuseIdentifier:方法，代码如下所示：

```

- (instancetype) initWithStyle: (UITableViewCellStyle) style reuseIdentifier:
(NSString *) reuseIdentifier
{
    self = [super initWithStyle: style reuseIdentifier: reuseIdentifier];
    if (self) {
        _webImageView = [[UIImageView alloc] initWithFrame: CGRectMake(0, 30,
self.frame.size.width, 100)];
        _titleLabel = [[UILabel alloc] initWithFrame: CGRectMake(0, 5, self.
frame.size.width, 21)];
        _titleLabel.backgroundColor = [UIColor clearColor];
        _titleLabel.textAlignment = NSTextAlignmentCenter;
        _titleLabel.textColor = [UIColor purpleColor];
        _titleLabel.font = [UIFont italicSystemFontOfSize: 17];
        _dateLabel = [[UILabel alloc] initWithFrame: CGRectMake(0, 140, self.
frame.size.width, 21)];
        _dateLabel.textAlignment = NSTextAlignmentRight;
        [self.contentView addSubview: _titleLabel];
        [self.contentView addSubview: _dateLabel];
        [self.contentView addSubview: _webImageView];
        self.selectionStyle = UITableViewCellSelectionStyleNone;
    }
    return self;
}

```


上面方法对 cell 以及其中控件进行了一些初始化的操作。

setContent:方法的实现如下所示:

```
-(void)setContent:(NSString *)content{
    _content=content;
    content = [content stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
    _webImageView.image = nil;
    [_webImageView sd_setImageWithURL:content completed:^(UIImage *image,
    NSError *error, SDImageCacheType cacheType, NSURL *imageURL) {
        _webImageView.frame=CGRectMake(0, 30, self.frame.size.width, self.frame.size.width/image.size.width*image.size.height);
        _dateLabel.frame=CGRectMake(0, _webImageView.frame.size.height+40, _dateLabel.frame.size.width, 21);
        if (_cellHeight!=_webImageView.frame.size.height+61) {
            _cellHeight = _webImageView.frame.size.height+61;
            NSNotification * noti = [NSNotification notificationWithName:@"sizeChange" object:nil userInfo:@{@"height":[NSString stringWithFormat:@"%f", _webImageView.frame.size.height+61],@"row":[NSString stringWithFormat:@"%d", _indexRow]}}];
            [[NSNotificationCenter defaultCenter]postNotification:noti];
        }
    }];
}
```

上面代码中 sd_setImageWithURL:completed:方法采用异步方式为 UIImageView 对象加载网络图片, 并且下载的图片数据会被缓存。在加载图片完后的 block 中, 进行 cell 的重新布局, 根据网络图片原始大小的比例进行 _webImageView 尺寸的重设, 最后使用 NotificationCenter 进行 cell 尺寸改变的通知, 在 UITableViewController 中监听这个通知来进行 cell 高度的重设。



提示

NSNotificationCenter 是系统提供的一个单例通知管理中心, 其使用通知的方法进行不同界面之间的传值十分方便。

ImageTableViewCell.m 中其他方法的实现代码如下:

```
-(CGFloat)getHeight{
    return _cellHeight;
}
-(void)setTitle:(NSString *)title{
    _title= title;
    _titleLabel.text = title;
}
-(void)setDate:(NSString *)date{
```

```

    _date= date;
    _dateLabel.text = date;
}

```

由于“趣图吧”界面需要异步加载大量的图片数据，图片的尺寸并不确定，这里采用的设计思路是当图片加载完成后发送消息通知，在 `UITableViewController` 中监听消息，进行界面的重新布局。这种方式在开发中并不常用，这种情况下，大多数开发者会采用 `autolayout`（自动布局）的方式来实现，关于 `autolayout` 的相关知识，会在后面章节进行介绍。

4.6.6 “趣图吧”界面的设计

在 `ImageViewController.m` 文件中引入头文件并进行一个宏定义，代码如下：

```

#import "MyNetWorking.h"
#import "RootDataModel.h"
#import "ImageTableViewCell.h"
#define IMAGEURL @"http://apis.baidu.com/showapi_open_bus/showapi_joke/joke_pic?page=%d"

```

在 `ImageViewController.m` 中声明如下变量，其意义与 `ViewController` 中相同。

```

@interface ImageViewController ()
{
    int _currentPage;
    NSMutableArray * _dataArray;
    UIButton * _moreBtn;
}
@end

```

在 `ImageViewController.m` 中的 `viewDidLoad` 方法中实现如下代码：

```

- (void)viewDidLoad {
    [super viewDidLoad];
    _currentPage = 1;
    self.title = @"趣图吧";
    [MyNetWorking sharedNetWorking].HTTPHeadersDic = @{@"apikey":@"c925fb
c1226c37b905a4d1e2a8cbbe99"};
    _dataArray = [[NSMutableArray alloc] init];
    _moreBtn = [UIButton buttonWithType:UIButtonTypeSystem];
    _moreBtn.frame=CGRectMake(0, 0, self.view.frame.size.width,30);
    [_moreBtn setTitle:@"加载更多~~" forState:UIControlStateNormal];
    [_moreBtn addTarget:self action:@selector(readMore) forControlEvents:
    UIControlEventTouchUpInside];
    [self getDataWithPage:_currentPage];
    [self creatRefresh];
}

```

```

[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(update:) name:@"sizeChange" object:nil];
self.tableView.tableFooterView = _moreBtn;
}

```

viewDidLoad 方法中除了进行一些初始化操作外还添加了一个通知的监听，当通知中心 **NSNotificationCenter** 发送 **sizeChange** 通知后会执行 **update:** 方法进行界面的更新。

update: 方法的实现如下：

```

-(void)update:(NSNotification *)noti{
    NSDictionary * dic = noti.userInfo;
    int row =[dic[@"row"] intValue];
    DataContentistModel * model = _dataArray[row];
    model.height = [dic[@"height"] floatValue];
    NSIndexPath *indexPath=[NSIndexPath indexPathForRow:row inSection:0];
    NSArray * cellArray = [self.tableView visibleCells];
    for (int i=0; i<cellArray.count; i++) {
        ImageTableViewCell * cell = cellArray[i];
        if (cell.indexRow==row) {
            [self.tableView reloadRowsAtIndexPaths:[NSArray arrayWithObject
s:indexPath,nil] withRowAnimation:UITableViewRowAnimationNone];
        }
    }
}

```

在上面的代码中，获取到通知对象传递进来的行数和行高，单独刷新 **UITableView** 的此行，由于 **UITableView** 的滑动是实时改变的，在进行刷新之前需要先判断这个 **cell** 是否可见，即是否正显示在屏幕上，如果显示正在进行刷新操作，**UITableView** 的 **visibleCells** 属性会返回所有可见的 **cell**。

实现 **creatRefresh** 与 **readMore** 方法的代码如下所示。

```

-(void)creatRefresh{
    self.refreshControl.tintColor = [UIColor greenColor];
    [self.refreshControl addTarget:self action:@selector(change:) forControlEvents:UIControlEventValueChanged];
}
-(void)readMore{
    [self getDataWithPage:_currentPage];
}

```

实现下拉刷新的逻辑方法 **change:** 如下所示。

```

-(void)change:(UIRefreshControl *)refreshControl{
    _moreBtn.enabled=NO;
}

```



```

[MyNetWorking getRequestWithURLString:[NSString stringWithFormat:IMAG
EURL,1] finish:^(BOOL success, NSData *data) {
    if (success) {
        NSDictionary * dataDic = [NSJSONSerialization JSONObjectWithDat
a:data options:NSJSONReadingMutableContainers error:nil];
        if (dataDic==nil) {
            //如果因为数据的问题不能解析, 跳过此页数据
        }else{
            NSDictionary * dic = dataDic[@"showapi_res_body"];
            RootDataModel * model = [[RootDataModel alloc] init];
            [model setValuesForKeysWithDictionary:dic];
            [_dataArray replaceObjectsInRange:NSMakeRange(0, model.cont
entlist.count) withObjectsFromArray:model.contentlist];
            [self.tableView reloadData];
        }
    }else{
        //进行网络问题的提示
        UIAlertController * alert = [UIAlertController alertControllerW
ithTitle:@"温馨提示" message:@"网络或服务异常" preferredStyle:UIAlertControlle
rStyleAlert];
        UIAlertAction * action = [UIAlertAction actionWithTitle:@"OK" s
tyle:UIAlertActionStyleCancel handler:nil];
        [alert addAction:action];
        [self presentViewController:alert animated:YES completion:nil];
    }
    _moreBtn.enabled=YES;
    [self.refreshControl endRefreshing];
}];
}

```

实现获取 API 服务数据的方法如下所示。

```

- (void)getDataWithPage:(int)page{
    [MyNetWorking getRequestWithURLString:[NSString stringWithFormat:IMAG
EURL,page] finish:^(BOOL success, NSData *data) {
        if (success) {
            _currentPage++;
            NSDictionary * dataDic = [NSJSONSerialization JSONObjectWithDat
a:data options:NSJSONReadingMutableContainers error:nil];
            if (dataDic==nil) {
                //如果因为数据的问题 不能解析 跳过此页数据
                [self getDataWithPage:_currentPage];
            }else{
                NSDictionary * dic = dataDic[@"showapi_res_body"];

```

```

        RootDataModel * model = [[RootDataModel alloc] init];
        [model setValuesForKeysWithDictionary:dic];
        [_dataArray addObjectsFromArray:model.contentlist];
        [self.tableView reloadData];
    }
} else {
    //进行网络问题的提示
    UIAlertController * alert = [UIAlertController alertControllerWithTitle:@"温馨提示" message:@"网络或服务异常" preferredStyle:UIAlertControllerStyleAlert];
    UIAlertAction * action = [UIAlertAction actionWithTitle:@"OK" style:UIAlertActionStyleCancel handler:nil];
    [alert addAction:action];
    [self presentViewController:alert animated:YES completion:nil];
}
}];
}
}

```

实现 UITableViewDelegate 与 UITableViewDataSource 协议中相关方法如下所示。

```

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:(NSInteger)section {
    return _dataArray.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    ImageTableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"ImageTableViewCell"];
    if (cell == nil) {
        cell = [[ImageTableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"ImageTableViewCell"];
    }
    cell.indexRow = (int)indexPath.row;
    DataContentListModel * model = _dataArray[indexPath.row];
    cell.title = model.title;
    cell.date = [NSString stringWithFormat:@"更新时间:%@", [model.ct substringToIndex:10]];
    if (![cell.content isEqualToString:model.img]) {
        cell.content = model.img;
    } else {
        model.height = [cell getHeight];
    }
}

```

```

    }
    return cell;
}

-(CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath{
    DataContentistModel * model = _dataArray[indexPath.row];
    if (model.height==0) {
        return 61;
    }
    return model.height;
}

```

运行工程，效果如图 4-31 所示。



图 4-31 “趣图吧”界面效果

最后，不要忘记配置应用图标、名称和启动页面。“笑一笑”休闲应用的核心功能到此就开发完成了，学习之余，娱乐一下吧！

第 5 章

音频、视频开发

在 App Store 中不乏有许多音乐播放器、视频播放器等优秀软件。iOS 开发框架中对音频和视频的开发提供了十分强大的支持。通过 `AVAudioPlayer` 类开发者可以十分便捷地进行音频相关需求的开发，通过 `MPMoviePlayerController` 类可以调用系统封装好的视频播放控件进行内嵌视频播放器的开发。本章在介绍音频、视频相关开发技术的基础上，将通过一款“音乐播放器”软件的实战演练使读者熟练运用这些 iOS 技术。

通过本章的学习，读者能够掌握：

1. iOS 音频播放技术。
2. iOS 视频播放技术。
3. iOS 后台播放音频并在桌面和锁屏界面显示音频信息。
4. 同步 LRC 歌词展示的音频播放。
5. iOS 画中画技术的使用。

5.1 iOS 音频开发基础——AVAudioPlayer 类的使用

AVAudioPlayer 是 iOS 中用来处理音频播放的一个类，其中提供的接口简单并且可以完成开发中大多数音频播放的需求。AVAudioPlayer 支持如下格式的音频文件播放：AAC、ALAC、HE_AAC、iLBC、IMA4、MP3。

5.1.1 使用 AVAudioPlayer 进行 MP3 音频文件的播放

使用 Xcode 创建一个名为 AVAudioPlayerTest 的工程，在工程的文件导航栏中单击鼠标右键，选择 Add Files To “AVAudioPlayerTest”...选项，如图 5-1 所示。

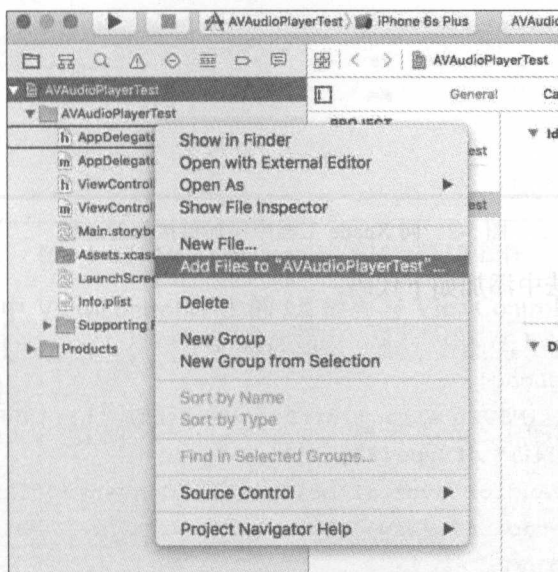


图 5-1 向 Xcode 工程中添加文件

选择一个 MP3 格式的文件进行添加，由于 AVAudioPlayer 是 AVFoundation.framework 框架中的类，因此需要在工程导入 AVFoundation.framework 这个系统框架。在 Xcode 开发工具的文件导航中选择工程文件，选择其中的 Build Phases 选项，单击 Link Binary With Libraries 栏目，再单击其中的加号添加链接库。操作过程如图 5-2 所示。

在 ViewController.m 文件中添加 AVFoundation.framework 的头文件：

```
#import <AVFoundation/AVFoundation.h>
```

在 ViewController.m 文件中声明一个 AVAudioPlayer 对象作为音频播放器，代码如下：

```
@interface ViewController ()
{
    AVAudioPlayer * _player;
```

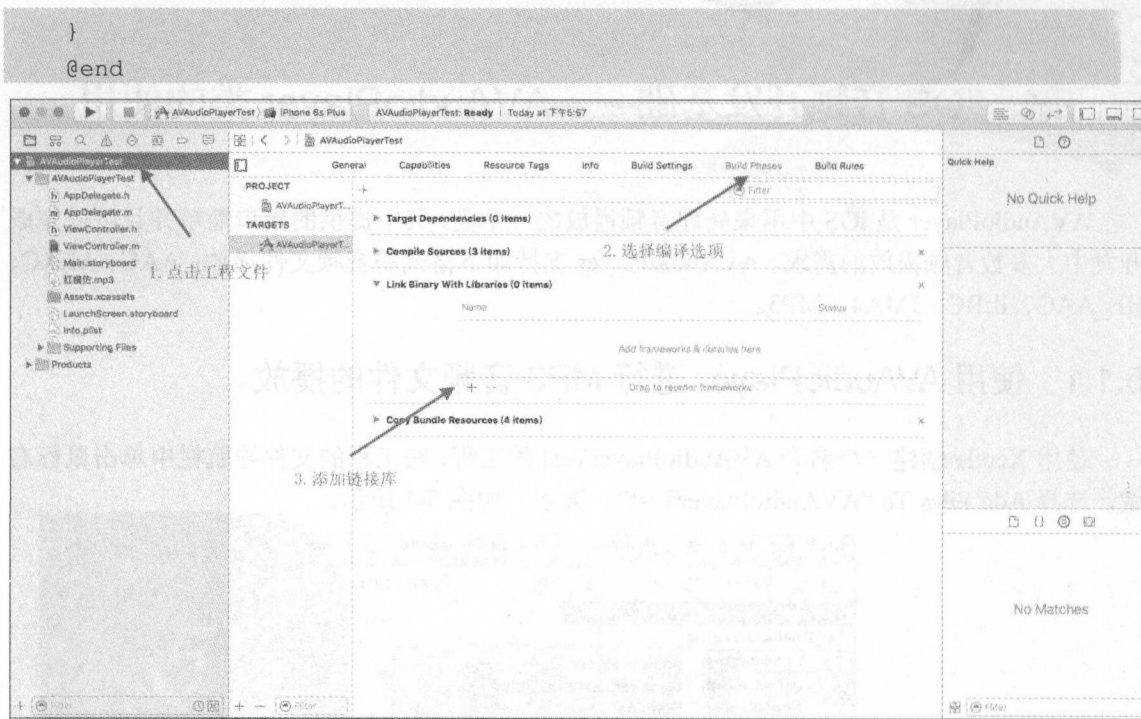


图 5-2 向 Xcode 工程中添加链接库的过程

在 viewDidLoad 方法中添加如下代码：

```

- (void)viewDidLoad {
    [super viewDidLoad];
    NSURL * url = [[NSURL alloc] initWithFileURLWithPath:[NSBundle mainBundle]
pathForResource:@"红模仿" ofType:@"mp3"]];
    _player = [[AVAudioPlayer alloc] initWithContentsOfURL:url error:nil];
    if ([_player prepareToPlay]) {
        [_player play];
    }
}

```

上面代码中，initWithFileURLWithPath:方法通过文件地址创建文件的 URL 路径对象，在工程中添加的文件地址可以通过[NSBundle mainBundle]pathForResource:ofType:方法来获取。pathForResource:ofType:方法中第 1 个参数是文件的名称，第 2 个参数是文件的格式类型，即后缀名。

AVAudioPlayer 对象通过 initWithContentsOfURL:error:方法来创建，这个方法需要传入音频文件的路径 URL 作为参数。

AVAudioPlayer 类的 prepareToPlay 方法用于做播放音频文件前的准备，在调用这个方法时会返回一个 BOOL 类型的返回值，这个值决定 AVAudioPlayer 对象准备工作是否成功完成，如果返回 YES，则可以调用 play 方法进行音频播放。

运行工程，打开音响，就可以听到歌曲了。

5.1.2 进行音频播放相关属性的控制

通常的音频播放软件都会支持暂停、续播、音量调节、声道调节、播放速度调节、播放时间监听等。在 AVAudioPlayerTest 工程中的 Main.storyboard 文件中添加一些简单的 UI 控件，使其看上去如图 5-3 所示。

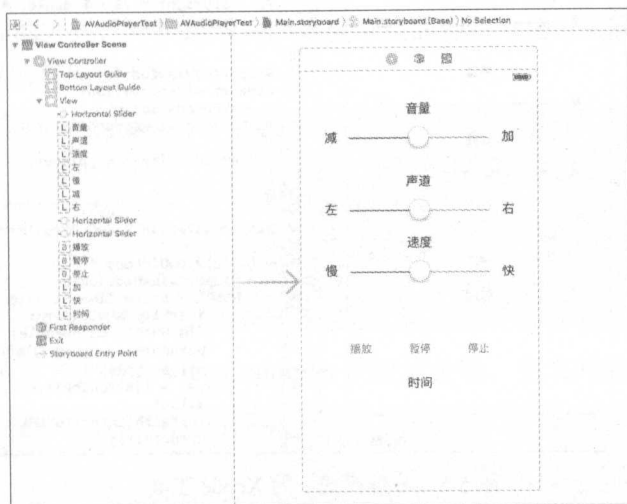


图 5-3 在 Main.storyboard 中添加一些 UI 控件

将 Main.storyboard 中 ViewController 上的 UI 控件与 ViewController 类进行关联，只需要关联和用户有交互或者 UI 展示上会变化的控件即可，就如上面的 3 个滑块控件、3 个按钮控件和最下面的时间标签控件。

选中 Main.storyboard 中当前的 ViewController，单击 Xcode 中的编辑助手按钮，如图 5-4 所示。

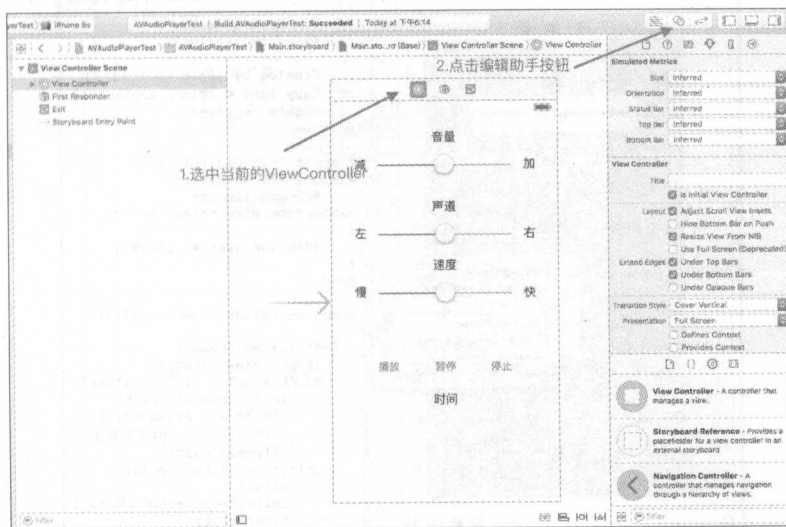


图 5-4 单击 Xcode 中的编辑助手按钮

此时 Xcode 开发工具会变成分屏模式，屏幕的左半部分是 storyboard 工具，右半部分是对应的类文件，如图 5-5 所示。

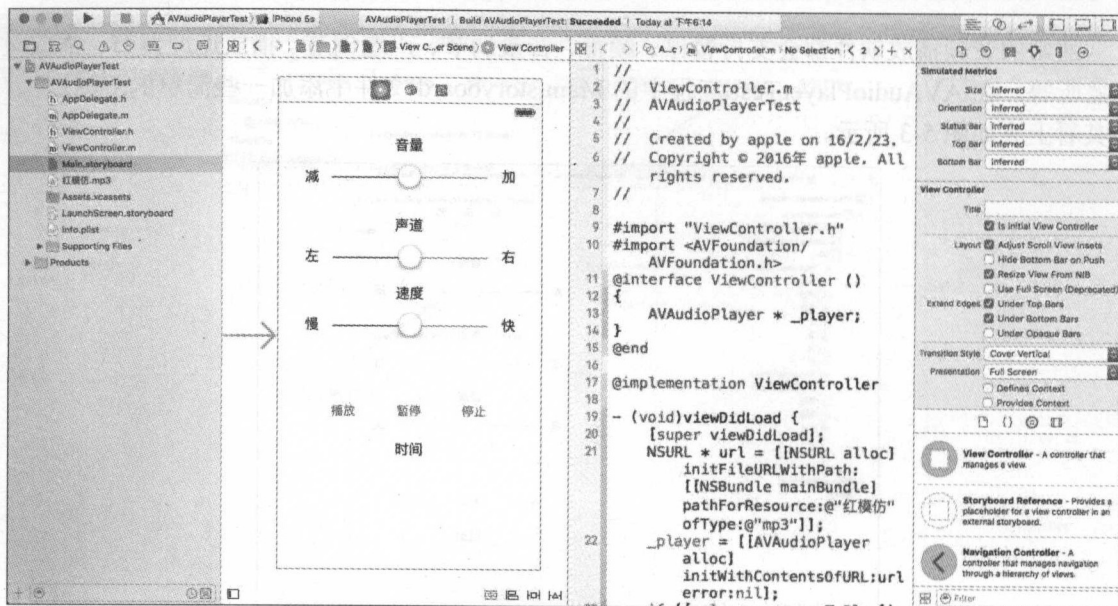


图 5-5 分屏模式下的 Xcode 工具

将 storyboard 中的控件与类文件进行关联有两种方式，一种是关联成属性，一种是关联成方法。任何控件都可以以属性的方式与类文件进行关联，但是只有可以进行用户交互的控件才用方法的方式与类关联。

用鼠标在左侧的 storyboard 中单击滑块控件，按住 control 键不放，将鼠标移动至右侧类文件的方法实现部分，即 @implementation 与 @end 之间松开鼠标，此时会出现 Xcode 如图 5-6 所示的配置窗口。

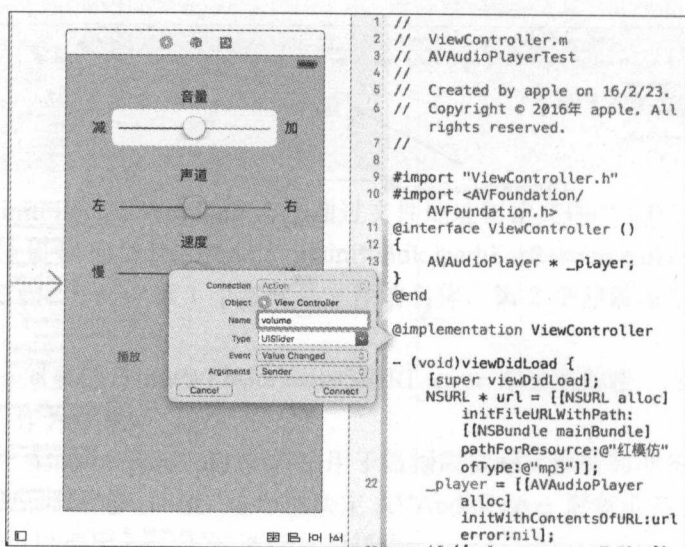


图 5-6 storyboard 中控件与类文件关联配置窗口

在图 5-6 中弹出的窗口中的 Name 项用于设置关联的方法名, Type 选项用于选择参数的类型, 这里可以选择 UISlider, Event 项设置触发方式, 因为 UISlider 只能触发 UIEventValueChanged 方式, 所以这里系统默认已经设置好, 开发者不能选择。设置完成后, 单击 Connect 按钮, Xcode 会自动在 ViewController.m 文件中创建一个名为 volume: 的函数, 当用户滑动音量下面的滑块控件时, volume: 方法会被调用, 开发者可以在其中进行音量改变的逻辑操作。

使用和上面同样的方式, 将声道滑块和速度滑块的触发方法也关联在 ViewController 类中, Xcode 自动生成的代码如下所示:

```
- (IBAction)volume:(UISlider *)sender {
}
- (IBAction)soundTrack:(UISlider *)sender {
}
- (IBAction)speed:(UISlider *)sender {
}
```

将 3 个按钮的触发方法也与 ViewController 类进行关联, 生成的触发方法如下所示。

```
- (IBAction)play:(UIButton *)sender {
}
- (IBAction)pause:(UIButton *)sender {
}
- (IBAction)stop:(UIButton *)sender {
}
```

界面最后的 UILabel 控件用于显示音频的总时长和已经播放的时长, 因其需要实时更新, 将此 UILabel 控件作为属性和 ViewController 类进行关联。和前面的方法关联类似, 在 storyboard 中选中控件按住 control 键不放, 将鼠标拖动至 ViewController.m 文件的 @interface 与 @end 之间, 如图 5-7 所示。

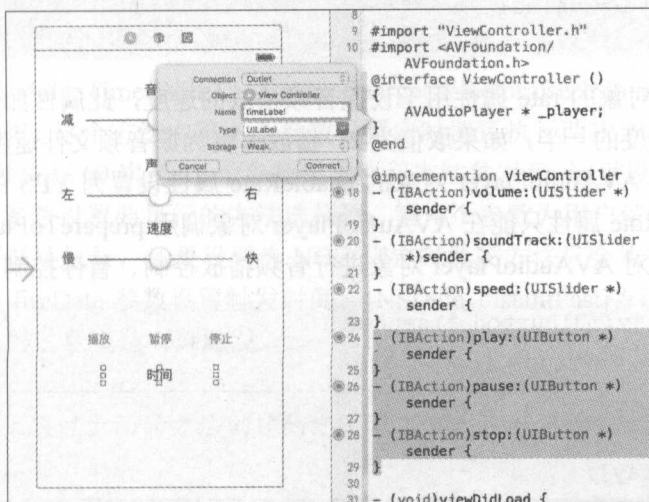


图 5-7 storyboard 中控件关联成类属性的配置窗口

上面进行关联时的配置窗口中，Name 项设置属性的名称，单击 Connect 按钮后，Xcode 会自动声明一个 timeLabel 属性，如下所示。

```
@interface ViewController ()
{
    AVAudioPlayer * _player;
    __weak IBOutlet UILabel *timeLabel;
}
@end
```

通过上面的操作，已经在 ViewController.m 中做好了属性与方法的关联，实现关联的方法如下所示：

```
- (IBAction)volume:(UISlider *)sender {
    float value = sender.value;
    _player.volume = value;
}
```

AVAudioPlayer 对象的 volume 属性用于设置音频播放的音量，这个属性的取值范围为 0~1 之间。

```
- (IBAction)soundTrack:(UISlider *)sender {
    float value = (sender.value-0.5)*2;
    _player.pan = value;
}
```

AVAudioPlayer 对象的 pan 属性设置声道的偏移，其取值范围为-1~1 之间，-1 为完全左声道，1 为完全右声道。

```
- (IBAction)speed:(UISlider *)sender {
    float value = sender.value+0.5;
    _player.rate = value;
}
```

AVAudioPlayer 对象的 rate 属性用于设置音频播放的速度，此属性如果取值为 0.5，播放速度为原音频文件速度的一半，如果取值为 2，播放速度为原音频文件速度的两倍。这里有一点需要注意，只有当 AVAudioPlayer 对象的 enableRate 属性设置为 YES 的时候，rate 属性才会生效，并且 enableRate 属性只能在 AVAudioPlayer 对象调用 prepareToPlay 方法之后设置。

下面 3 个方法是对 AVAudioPlayer 对象进行音频播放控制、暂停控制和停止控制的方法。

```
- (IBAction)play:(UIButton *)sender {
    if (_player.playing) {
        return;
    }
    [_player play];
}

- (IBAction)pause:(UIButton *)sender {
```

```

    if (_player.isPlaying) {
        [_player pause];
    }
}

- (IBAction)stop:(UIButton *)sender {
    if ([_player isPlaying]) {
        [_player stop];
    }
}
}

```



提示

AVAudioPlayer 类对象的 pause 方法和 stop 方法的区别在于 pause 方法只是做暂停播放的操作,在再次播放时 AVAudioPlayer 对象不需要再做准备工作; stop 方法则是停止操作,当 AVAudioPlayer 再次播放时,需要重新做准备工作,除了上面介绍的常用方法外, AVAudioPlayer 还有一个 numberOfLoops 的属性用来设置音频文件播放的循环次数。

还有一个音频播放时间的 UILabel 控件没有做处理,为了实时地对 UILabel 控件上的信息进行更新,在 ViewController.m 文件中声明一个定时器 NSTimer 对象。

```

{
    AVAudioPlayer * _player;
    __weak IBOutlet UILabel *timeLabel;
    NSTimer * _timer;
}

```

在 viewDidLoad 方法中对 NSTimer 对象进行创建操作,代码如下:

```

_timer = [NSTimer scheduledTimerWithTimeInterval:1/60 target:self selector:
:@selector(updateProgress) userInfo:nil repeats:YES];
_timer.fireDate = [NSDate distantPast];

```

上面代码中, scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:方法用于定时器的创建和初始化设置,这个方法中第 1 个参数设置定时器每执行两次方法间的时间间隔,这里设置为 1/60,即每秒执行 60 次;第 2 个参数设置执行方法的对象,这里设置为 ViewController 类对象本身;第 3 个参数设置要执行的方法选择器;第 4 个参数为用户信息,可以设置为 nil;最后一个参数为是否循环执行,如果设置为 NO,则定时器的方法只会执行一次。

NSTimer 对象的 fireDate 参数设置触发时间,[NSDate distantPast]方法将创建一个很久之前的时间,对应定时器的意义是立刻触发。



提示

实际上通过上面的方法创建的定时器不需要设置 fireDate 参数也会自动执行触发。

定时器的触发方法 updateProgress 方法的实现如下:

```
-(void)updateProgress{
    int current = _player.currentTime;
    int duration = _player.duration;
    timeLabel.text = [NSString stringWithFormat:@"%02d:%02d:%02d",current/60,current%60,duration/60,duration%60];
}
```

运行工程，效果如图 5-8 所示。



图 5-8 音频播放属性控制界面

5.1.3 后台播放音频及用户交互的优化

对于一款音频播放软件，后台播放是必备的功能，任何一个音乐播放器都不可能要求用户始终不锁屏的盯着屏幕看。使前面章节中创建的 AVAudioPlayerTest 工程支持后台音频播放十分简单，只需要在工程的配置文件 Info.plist 文件中添加一个键值即可。

打开 AVAudioPlayer 工程，单击进入 Info.plist 文件，向其中添加如图 5-9 所示的键值。

Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
▼ Required background modes	Array	(1 item)
item 0	String	App plays audio or streams audio/video using AirPlay
Localization native development re...	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
► Required device capabilities	Array	(1 item)
► Supported interface orientations	Array	(3 items)

图 5-9 向 info.plist 文件中添加支持后台音频播放的键值

设置完 Info.plist 文件后,在 AppDelegate.m 文件的 application:FinishLaunchingWithOptions: 方法中添加如下代码:

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    AVAudioSession *session = [AVAudioSession sharedInstance];
    [session setActive:YES error:nil];
    [session setCategory:AVAudioSessionCategoryPlayback error:nil];
    [[UIApplication sharedApplication] beginReceivingRemoteControlEvents];

    return YES;
}
```

上面的代码做了音频后台播放的基础配置并监听了音频播放的后台用户交互。

此时再次运行工程,按下 Home 键进入后台,可以发现音频还在播放并没有停止。

在 iPhone 主界面从下往上拉可以拉出一个抽屉视图。对于音频播放类的软件,如果监听了用户之后后台才做,即在程序中调用了如下代码,那么 iPhone 主界面的抽屉视图如图 5-10 所示。

```
-(void)remoteControlReceivedWithEvent:(UIEvent *)event{
    if (event.type==UIEventTypeRemoteControl) {
        switch (event.subtype) {
            //单击播放按钮或者耳机线控中间的那个按钮
            case UIEventSubtypeRemoteControlPlay:
            {
                NSLog(@"play");
            }
            break;
            //单击暂停与播放切换按钮即抽屉中间的按钮
            case UIEventSubtypeRemoteControlTogglePlayPause:
            {
                NSLog(@"pause-play");
            }
            break;
            //单击下一曲或者耳机线控中间的按钮连击两下
            case UIEventSubtypeRemoteControlNextTrack:
            {
                NSLog(@"next");
            }
            break;
            //单击上一曲按钮或者耳机线控中间的按钮三下
            case UIEventSubtypeRemoteControlPreviousTrack:
            {
                NSLog(@"previous");
            }
        }
    }
}
```

```

        break;
    default:
        break;
    }
}
}

```

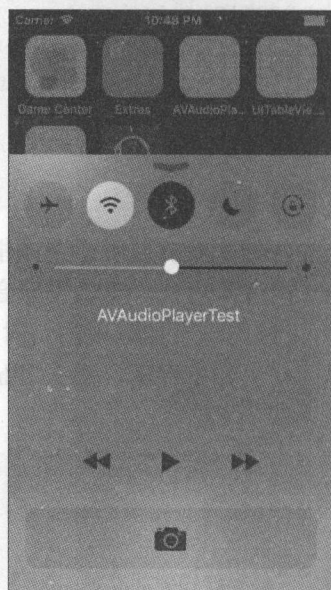


图 5-10 iPhone 的上拉抽屉视图

在图 5-10 中可以看出，抽屉视图中显示了所运行的音频播放软件的名称，即 AVAudioPlayerTest，并显示了 3 个按钮，一个播放按钮、一个上一曲按钮和一个下一曲按钮。如果用户现在单击这些按钮，其实并没有什么效果，开发者需要在 AVAudioPlayerTest 工程的 AppDelegate.m 文件中实现一个函数来处理监听到的用户交互事件，参见上述代码：

上面代码列举的是一些常用的交互枚举值，再次运行工程单击上拉抽屉中的按钮或者在真机上通过耳机进行线控操作，可以看到 Xcode 的调试区打印出对应的 Log 信息。

iOS 的音频系统除了可以在后台接收用户交互事件外，还可以在上拉抽屉与锁屏界面显示当前播放的音频的相关信息。这些信息的设置通过字典中的键值对来完成。例如首先在工程中 ViewController.m 文件引入如下头文件：

```
#import <MediaPlayer/MediaPlayer.h>
```

在 ViewDidLoad 方法的最后添加如下代码：

```

//设置歌曲题目
[dict setObject:@"牛仔很忙" forKey:MPMediaItemPropertyTitle];
//设置歌手名
[dict setObject:@"周杰伦" forKey:MPMediaItemPropertyArtist];
//设置专辑名
[dict setObject:@"我很忙" forKey:MPMediaItemPropertyAlbumTitle];

```

```

//设置显示的图片
UIImage *newImage = [UIImage imageNamed:@"我很忙.jpg"];
[dict setObject:[MPMediaItemArtwork alloc] initWithImage:newImage]
    forKey:MPMediaItemPropertyArtwork];
//设置歌曲时长
[dict setObject:[NSNumber numberWithInt:300] forKey:MPMediaItemPro
pertyPlaybackDuration];
//设置已经播放时长
[dict setObject:[NSNumber numberWithInt:150] forKey:MPNowPlayingIn
foPropertyElapsedPlaybackTime];
//更新字典
[[MPNowPlayingInfoCenter defaultCenter] setNowPlayingInfo:dict];

```

上面代码中,MPMediaItemPropertyTitle 键值对应歌曲的名称;MPMediaItemPropertyArtist 键值对应歌手的名称;MPMediaItemPropertyAlbumTitle 键值对应专辑名;MPMediaItemPropertybArtwork 键值对应要显示的图片,必须设置为 UIImage 对象;MPMediaItemProperty- PlaybackDuration 键值对应音频的总时长;MPNowPlayingInfoPropertyElapsedPlaybackTime 键值对应音频已经播放的时长。最后需要调用 [[MPNowPlayingInfoCenter defaultCenter]setNowPlayingInfo:dic]方法来使用新的配置字典对展示的信息进行重设。

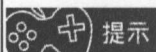
添加了上面的代码后,再次运行工程,可以看到如图 5-11 与图 5-12 所示的界面模样。



图 5-11 配置了音频信息的上拉抽屉界面



图 5-12 配置了音频信息的锁屏界面



提示

1. 模拟器锁屏的方法是 command+L 键。
2. 进行后台用户交互的上拉抽屉和锁屏界面的开发时,最好使用真机进行测试,在模拟器上会出现无法显示和按钮交互逻辑错乱等问题,如果读者遇到,不必惊慌。

5.2 iOS 视频开发基础

在 iOS8 系统之前，MPMoviePlayerController 类专门负责视频播放功能的开发，其支持远程视频和本地视频两种方式。使用 MPMoviePlayerController 和 MPMoviePlayerViewController 开发者可以十分轻松地将视频嵌入应用程序中。

5.2.1 使用 MPMoviePlayerController 向应用中嵌入视频模块

MPMoviePlayerController 除了支持视频播放，其中还继承了一些例如暂停、进度、全屏等播放器功能。可以理解其为视频播放视图和视频播放逻辑控制器的结合。

使用 Xcode 创建一个名为 MPMoviePlayerTest 的工程，向工程中添加一个 MP4 格式的视频文件，在 ViewController.m 文件中引入 MediaPlayer 框架，代码如下所示：

```
#import <MediaPlayer/MediaPlayer.h>
```

在 ViewController.m 文件中声明一个 MPMoviePlayerController 播放器对象，代码如下所示：

```
@interface ViewController ()
{
    MPMoviePlayerController * _movieController;
}
```

在 viewDidLoad 方法中添加如下代码，对 MPMoviePlayerController 对象进行创建和初始化。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSString * path = [[NSBundle mainBundle] pathForResource:@"iphone" ofType:@"mp4"];
    NSURL * url = [NSURL fileURLWithPath:path];
    _movieController = [[MPMoviePlayerController alloc] initWithContentURL:url];
    _movieController.view.frame=CGRectMake(0, 0, 320, 300);
    [self.view addSubview:_movieController.view];
    [_movieController play];
}
```

运行工程，视频播放器效果如图 5-13 所示。

上面 viewDidLoad 方法中的代码，通过本地视频文件的 URL 创建了 MPMoviePlayerController 对象。MPMoviePlayerController 对象中的 view 属性是播放器视图，在使用时，应将其位置尺寸进行设置并通过 addSubview: 方法添加在当前视图上。MPMoviePlayerController 对象调用 play 方法来进行视频的播放操作。

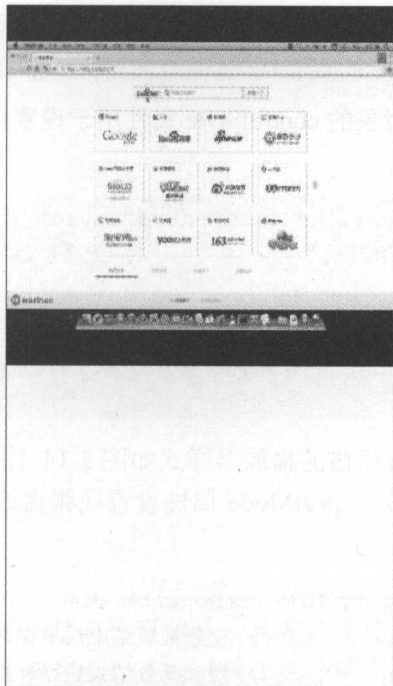


图 5-13 视频播放界面

5.2.2 MPMoviePlayerController 常用属性与方法解析

MPMoviePlayerController 对象有一个名为 `playbackState` 的属性, 通过这个属性开发者可以获取到 MPMoviePlayerController 对象的当前播放状态。`playbackState` 属性是一个 MPMoviePlaybackState 类型的枚举, 枚举值意义如下所示:

```
typedef NS_ENUM(NSInteger, MPMoviePlaybackState) {
    MPMoviePlaybackStateStopped,           // 视频播放器处于停止状态
    MPMoviePlaybackStatePlaying,           // 视频播放器处于播放状态
    MPMoviePlaybackStatePaused,            // 视频播放器处于暂停状态
    MPMoviePlaybackStateInterrupted,       // 视频播放器处于中断状态
    MPMoviePlaybackStateSeekingForward,    // 视频播放器处于快进状态
    MPMoviePlaybackStateSeekingBackward   // 视频播放器处于快退状态
};
```

MPMoviePlayerController 对象的 `loadState` 属性可以获取当前视频数据的加载状态, 枚举意义如下所示:

```
typedef NS_OPTIONS(NSUInteger, MPMovieLoadState) {
    MPMovieLoadStateUnknown      = 0, // 状态未知
    MPMovieLoadStatePlayable     = 1 << 0, // 缓存数据足够开始播放, 但是视频并没有缓存完全
    MPMovieLoadStatePlaythroughOK = 1 << 1, // 已经缓存完成, 如果设置了自动播放, 这时会自动播放
};
```

```
MPMovieLoadStateStalled = 1 << 2, //数据缓存已经停止, 播放将暂停
};
```

MPMoviePlayerController 对象的 controlStyle 属性用于设置播放器的风格, 其枚举意义如下所示:

```
typedef NS_ENUM(NSInteger, MPMovieControlStyle) {
    MPMovieControlStyleNone,           // 无播放器
    MPMovieControlStyleEmbedded,       // 嵌入式风格
    MPMovieControlStyleFullscreen,     // 充满视图风格
    MPMovieControlStyleDefault = MPMovieControlStyleEmbedded //默认风格
};
```

各种 MPMovieControlStyle 风格的播放器样式如图 5-14~图 5-16 所示。

MPMoviePlayerController 的 repeatMode 属性设置视频播放的循环模式, 枚举值及其意义如下所示:

```
typedef NS_ENUM(NSInteger, MPMovieRepeatMode) {
    MPMovieRepeatModeNone,             //视频播放结束后不循环播放
    MPMovieRepeatModeOne                //视频播放结束后循环播放
};
```

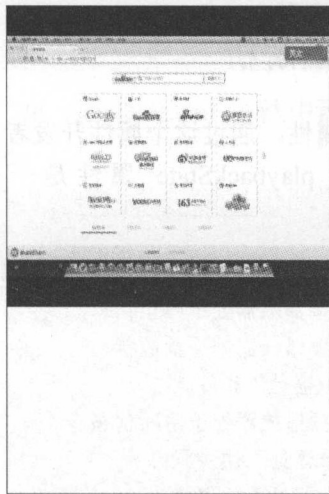


图 5-14 MPMovieControlStyleNone

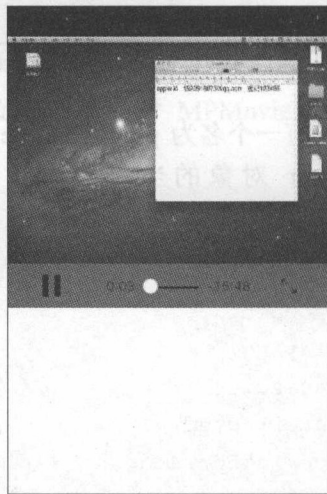


图 5-15 MPMovieControlStyleEmbedded

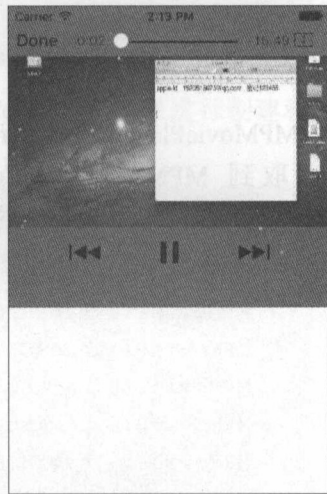


图 5-16 MPMovieControlStyleFullscreen

除了上面介绍的枚举属性外, MPMoviePlayerController 中关于播放控制的相关方法如下所示:

```
//调用这个方法进行播放视频的准备工作
- (void)prepareToPlay;
//获取播放器的准备工作是否就绪
@property(nonatomic, readonly) BOOL isPreparedToPlay;
//调用此方法进行视频的播放
```



```

- (void)play;
//调用此方法进行视频播放的暂停操作
- (void)pause;
//调用此方法停止视频播放
- (void)stop;
//当前视频已播放的时间
@property(nonatomic) NSTimeInterval currentTime;
//当前视频的播放速度
@property(nonatomic) float currentPlaybackRate;
//调用此方法进行快进操作
- (void)beginSeekingForward;
//调用此方法进行快退操作
- (void)beginSeekingBackward;
//调用此方法结束快进或者快退操作
- (void)endSeeking;

```

**提示**

关于视频播放的快进与快退操作，只有在进行视频文件的播放时才有效，这些方法对于网络视频数据流的播放情况并没有效果。

MPMoviePlayerController 中还有一些属性用于获取视频数据相关信息，如下所示：

```

//获取媒体的类型
@property(nonatomic, readonly) MPMovieMediaTypeMask movieMediaTypes;
//获取数据的类型
@property(nonatomic) MPMovieSourceType movieSourceType;
//获取数据的时长
@property(nonatomic, readonly) NSTimeInterval duration;
//获取可播放部分的时长
@property(nonatomic, readonly) NSTimeInterval playableDuration;
//获取原始尺寸大小
@property(nonatomic, readonly) CGSize naturalSize;
//设置启示播放的位置
@property(nonatomic) NSTimeInterval initialPlaybackTime;
//设置结束播放的位置
@property(nonatomic) NSTimeInterval endPlaybackTime;

```

某些功能强大的视频播放器是允许用户查看某一时刻的视频缩略图的，也可以提供给用户在观看视频当中进行截图的功能。通过 MPMoviePlayerController 开发者也可以十分方便地实现上述功能，MPMoviePlayerController 提供了视频文件中某一时刻的截图。

将 MPMoviePlayerTest 工程的 ViewController.m 文件中的 viewDidLoad 方法修改如下：

```

- (void)viewDidLoad {
    [super viewDidLoad];

```

```

NSString * path = [[NSBundle mainBundle]pathForResource:@"iphone" ofType:@"mp4"];
NSURL * url = [NSURL fileURLWithPath:path];
_movieController = [[MPMoviePlayerController alloc] initWithContentURL:url];
_movieController.controlStyle = MPMovieControlStyleFullscreen;
_movieController.view.frame=CGRectMake(0, 0, 320, 300);
[self.view addSubview:_movieController.view];
UIImage * image = [_movieController thumbnailImageAtTime:100 timeOption:MPMovieTimeOptionNearestKeyFrame];
UIImageView * imageView = [[UIImageView alloc] initWithFrame:CGRectMake(0, 350, 320, 200)];
imageView.image = image;
[self.view addSubview:imageView];
[_movieController play];
}

```

在上面的代码中，`thumbnailImageAtTime:timeOption:`方法用于截取视频中某一时刻的截图，这个方法中第 1 个参数设置将要截取的时间点，第 2 个参数设置截取的时间点设置选项，其可选择的枚举意义如下：

```

typedef NS_ENUM(NSInteger, MPMovieTimeOption) {
    MPMovieTimeOptionNearestKeyFrame, //截取所设置的时间点附近的关键帧作为截图
    MPMovieTimeOptionExact //截取所设置的时间点的精确帧作为截图
};

```

运行工程，效果如图 5-17 所示。



图 5-17 使用 MPMoviePlayerController 进行视频截图

5.3 视频播放器视图控制器——MPMoviePlayerViewController

在 5.2 节中,介绍了视频播放器控件 MPMoviePlayerController 类,这个类是单纯的视频播放视图和视频播放控制控件的组合。在 MediaPlayer 框架中还提供给开发者一个类:MPMoviePlayerViewController。MPMoviePlayerViewController 类将 MPMoviePlayerController 控件和视图控制器集成在了一起,MPMoviePlayerViewController 类继承自 UIViewController,并且其内部自带一个 MPMoviePlayerController 控件。

使用 Xcode 创建一个名为 MPMoviePlayerViewControllerTest 的工程,向其中添加一个 MP4 格式的视频文件,并在 Main.storyboard 文件中的视图控制器上添加一个 UIButton 控件,如图 5-18 所示。

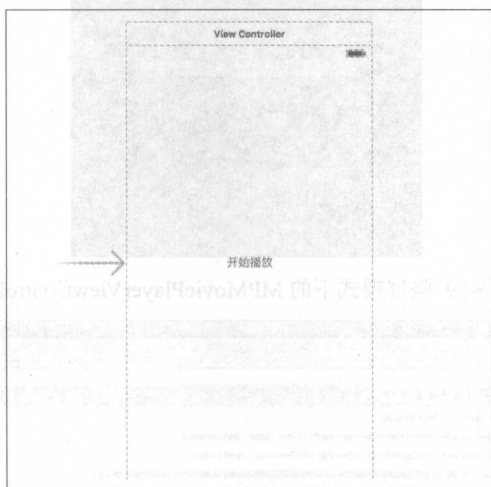


图 5-18 向 storyboard 文件的 ViewController 中添加一个按钮控件

将 storyboard 中 ViewController 上的按钮控件的触发方法与 ViewController.m 文件进行关联,方法名取为 playMovie,在 ViewController.m 文件中引入 MediaPlayer 框架的头文件,代码如下:

```
#import <MediaPlayer/MediaPlayer.h>
```

在 playMovie:方法中添加如下代码:

```
- (IBAction)playMovie:(id)sender {
    NSString * path = [[NSBundle mainBundle]pathForResource:@"iphone" ofType:@"mp4"];
    NSURL * url = [NSURL fileURLWithPath:path];
    MPMoviePlayerViewController * controller = [[MPMoviePlayerViewController alloc] initWithContentURL:url];
    [self presentMoviePlayerViewControllerAnimated:controller];
}
```


上面代码中，通过 `initWithContentURL:` 方法进行 `MPMoviePlayerViewController` 对象的初始化，当 `MPMoviePlayerViewController` 视图控制器对象被 `present` 的时候，视频会自动播放，效果如图 5-19 与图 5-20 所示。

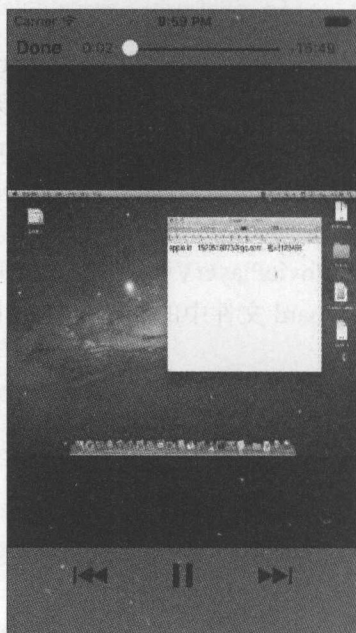


图 5-19 竖屏模式下的 `MPMoviePlayerViewController`

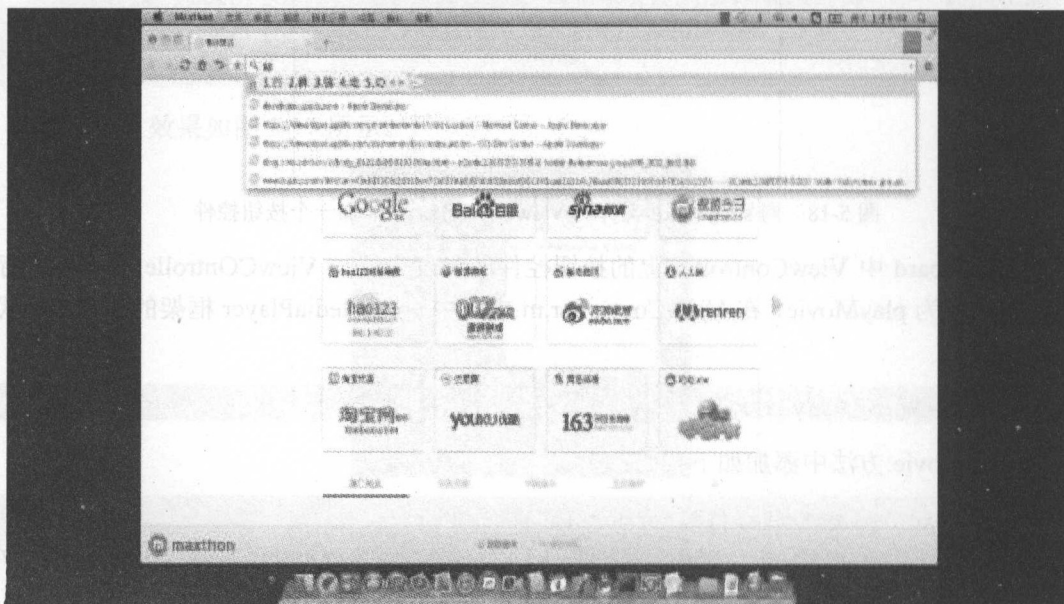


图 5-20 横屏模式下的 `MPMoviePlayerViewController`

当 `MPMoviePlayerViewController` 中的视频播放完毕或者单击屏幕左上角的 `Done` 按钮之后，`MPMoviePlayerViewController` 视图控制器会自动执行 `dismiss` 方法返回原先的视图控制器并释放所占据的内存空间。

5.4 AVPlayerViewController 视频播放框架与画中画开发技术

AVPlayerViewController 是 iOS8 之后引入的新的视频播放类, iOS 9 版本后 iPad 中引入了画中画技术, AVPlayerViewController 可以很好地帮助开发者进行画中画的开发, 并且在 iOS 9 中 MPMoviePlayerViewController 及其相关类也已完全被弃用。

5.4.1 使用 AVPlayerViewController 进行视频播放

使用 Xcode 创建一个名为 AVPlayerTest 的工程, 在 Main.storyboard 文件 ViewController 中拉入一个按钮控件, 如图 5-21 所示。

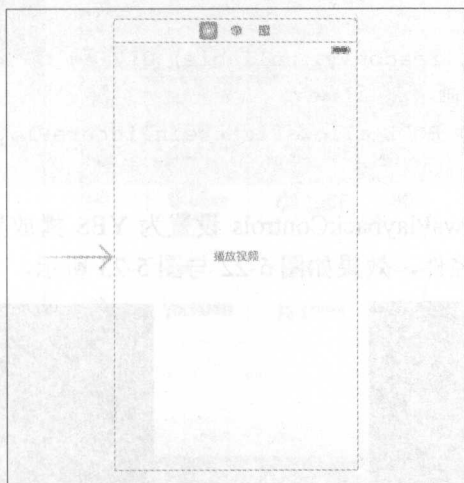


图 5-21 在 Main.storyboard 中添加一个按钮控件

将 Main.storyboard 文件中的按钮的触发方法与 ViewController.m 文件进行关联, 将方法命名为 playMovie。在 ViewController.m 文件中引入如下头文件:

```
#import <AVKit/AVKit.h>
#import <AVFoundation/AVFoundation.h>
```

playMovie 方法的实现如下所示:

```
-(IBAction)playMovie:(id)sender {
    NSString * path = [[NSBundle mainBundle]pathForResource:@"iphone" ofType:@"mp4"];
    NSURL *url = [NSURL URLWithString:path];
    AVPlayerViewController * play = [[AVPlayerViewController alloc] init];
    play.player = [[AVPlayer alloc] initWithURL:url];
    [self presentViewController:play animated:YES completion:nil];
}
```

AVPlayerViewController 对象中有一个 player 属性，这个属性是 AVPlayer 类型的对象，AVPlayer 是 AVFoundation 框架的播放控制类，其不仅可以播放视频，还可以播放音频。

AVPlayerViewController 中还有以下属性提供给开发者进行视频播放器属性的自定义。

```
//设置是否显示播放器控制控件
@property (nonatomic) BOOL showsPlaybackControls;

//设置播放区域拉伸模式
@property (nonatomic, copy) NSString *videoGravity;
//获取播放器是否已准备好播放
@property (nonatomic, readonly, getter = isReadyForDisplay) BOOL readyForDisplay;
//获取视频播放区域的尺寸
@property (nonatomic, readonly) CGRect videoBounds;
//播放器背景视图
@property (nonatomic, readonly, nullable) UIView *contentOverlayView;
//设置播放器是否支持画中画
@property (nonatomic) BOOL allowsPictureInPicturePlayback NS_AVAILABLE_IOS(9_0);
```

上面列出的属性中，showsPlaybackControls 设置为 YES 播放界面会带一个播放控制器控件，设置为 NO 则不显示此控件，效果如图 5-22 与图 5-23 所示。

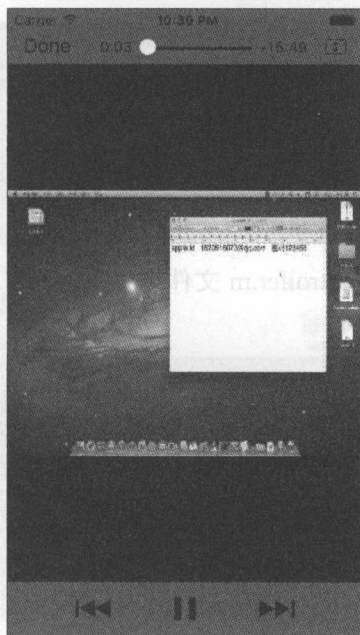


图 5-22 显示播放控制控件的视频播放界面

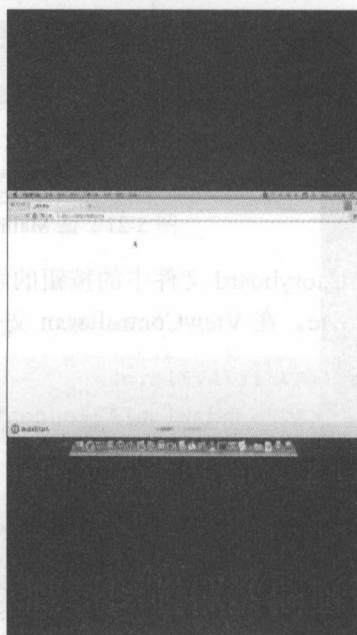


图 5-23 不显示播放控制控件的视频播放界面

videoGravity 属性用于设置播放区域的拉伸模式，支持的常亮字符串及意义如下：

- AVLayerVideoGravityResizeAspect 不拉伸比例，以播放区域宽高中大的值充满为标准

- AVLayerVideoGravityResizeAspectFill 不拉伸比例,以播放区域宽高中小值充满为标准
- AVLayerVideoGravityResize 进行比例拉伸充满界面

如果开发者不设置 videoGravity 的值,则默认是 AVLayerVideoGravityResizeAspect 的拉伸模式。

上面 3 种拉伸模式效果如图 5-24~图 2-26 所示。

contentOverlayView 属性是播放器的内容视图,如果开发者需要自定义一些播放器控件,可以将其添加在 contentOverlayView 上。

allowsPictureInPicturePlayback 属性设置是否支持画中画播放,在 iOS 9 版本之后可用并且默认是支持的。

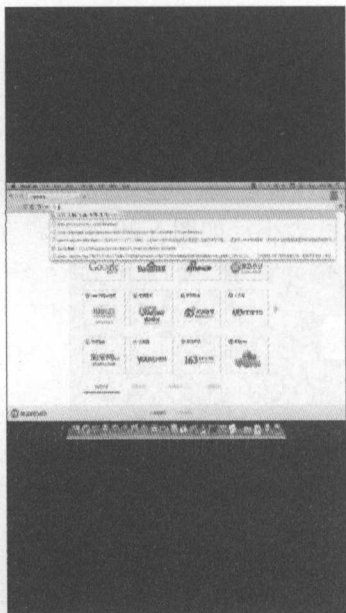


图 5-24 AVLayerVideoGravity-ResizeAspect



图 5-25 AVLayerVideoGravity-ResizeAspectFill

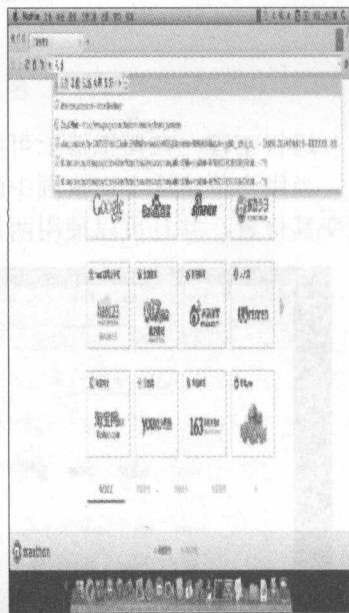


图 5-26 AVLayerVideoGravity-Resize

5.4.2 iPad 的画中画播放技术

画中画技术是 iPad 中多任务的一种,其允许用户将当前视频播放界面缩放在屏幕一角,在用户浏览应用程序中其他界面或者直接按 Home 回到主页面甚至切换到其他应用程序时,缩小的视频窗口可以在 iPad 屏幕上继续播放。这种画中画的播放方式极大地优化了用户体验,是 iOS 系统的一大亮点。

目前支持画中画技术的 iOS 设备有 iPad Air、iPad Air2、iPad mini2、iPad mini3 及更新设备,且系统的版本不低于 iOS 9。

将 Xcode 运行的模拟器选择为 iPad Air,如图 5-27 所示。

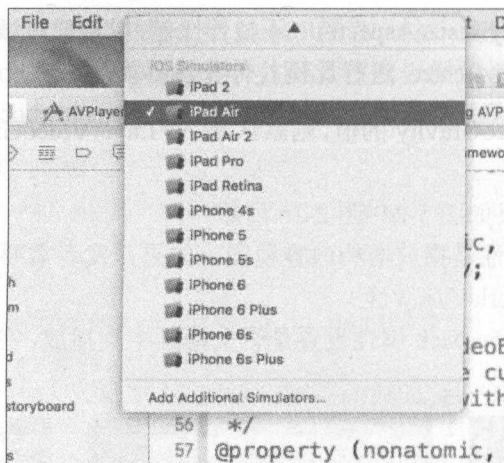


图 5-27 将 Xcode 的模拟器选择为 iPad Air

运行工程，在模拟器上可以看到播放界面有一个画中画切换的按钮，如图 5-28 所示。

当用户单击这个切换画中画按钮时，当前播放界面会被缩放至屏幕一角，可以通过拖动来改变其位置，并且可以使用两指捏合的手势来改变播放窗口的尺寸大小，如图 5-29 所示。

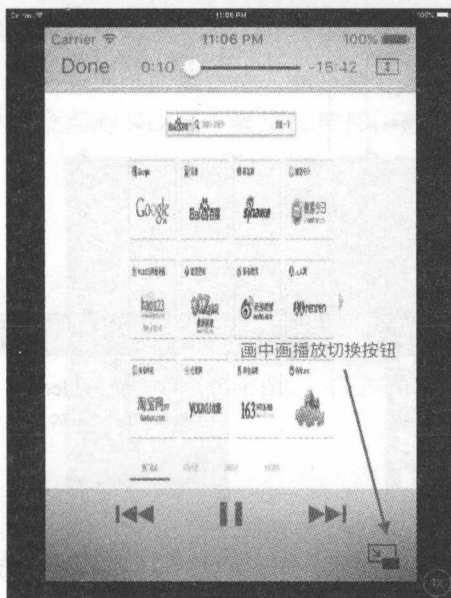


图 5-28 画中画切换按钮

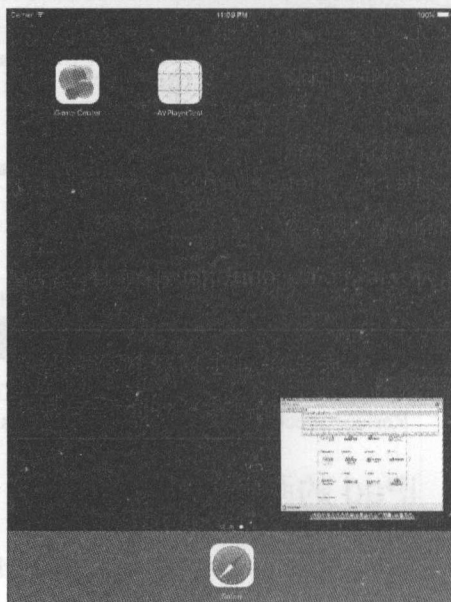


图 5-29 视频画中画播放模式

对于画中画的相关操作，开发者可以在程序中使用代理方法进行监听，AVPlayerViewController 对象中有 delegate 这样一个属性，在 ViewController.m 中遵守如下协议：

```
@interface ViewController ()<AVPlayerViewControllerDelegate>
@end
```

开发者通过 AVPlayerViewControllerDelegate 协议中约定的如下方法来进行用户画中画操作的监听。

```

//将要进入画中画模式时系统调用的方法
- (void)playerViewControllerWillStartPictureInPicture:(AVPlayerViewController *)playerViewController{
}
//已经进入画中画模式时系统调用的方法
- (void)playerViewControllerDidStartPictureInPicture:(AVPlayerViewController *)playerViewController{
}
//进入画中画模式失败系统调用的方法
- (void)playerViewController:(AVPlayerViewController *)playerViewController failedToStartPictureInPictureWithError:(NSError *)error{
}
//将要结束画中画模式系统调用的方法
- (void)playerViewControllerWillStopPictureInPicture:(AVPlayerViewController *)playerViewController{
}
//已经结束画中画模式时系统调用的方法
- (void)playerViewControllerDidStopPictureInPicture:(AVPlayerViewController *)playerViewController{
}
//用户操作画中画视图中的还原按钮时系统调用的方法
- (void)playerViewController:(AVPlayerViewController *)playerViewController restoreUserInterfaceForPictureInPictureStopWithCompletionHandler:(void (^)(BOOL restored))completionHandler{
}
//进入画中画模式后 是否自动将当前播放视图控制器 dismiss 掉
- (BOOL)playerViewControllerShouldAutomaticallyDismissAtPictureInPictureStart:(AVPlayerViewController *)playerViewController{
    return YES;
}

```

在上面的协议方法中，只有 `playerViewControllerShouldAutomaticallyDismissAtPictureInPictureStart` 方法比较特殊，这个方法有一个 `BOOL` 类型的返回值。当返回 `YES` 时，在进入画中画模式后，当前的视频播放视图控制器会自动调用 `dismiss` 方法来返回到上级视图控制器，如果返回 `NO`，播放窗口缩小后，界面依然会停留在当前的视频播放视图控制器。



提示

`AVPlayerViewControllerDelegate` 协议中约定的方法只能在 iOS 9 系统之后可用。

5.5 实战：“天后王菲”音频播放器的开发

在本小节中读者将通过开发一款音乐播放器类应用程序学习到更多技巧，通过歌词同步引擎的开发，读者将初步掌握 iOS 中数据解析的相关方法。

5.5.1 工程搭建与 LRC 歌词文件简介

一款流行的音乐播放器软件，在播放音乐时同步显示歌词是必备的功能，其实这些播放器软件都实现了一个歌词解析引擎，通过歌词解析引擎将 LRC 文件解析为程序中需要的歌词对象。

LRC 是 Lyric 单词的缩写，是音频同步歌词文件的一种协议格式。LRC 文件中通过标签的形式将歌曲的专辑、歌手、每个时间点对应的歌词记录其中，音频播放器通过解析这些数据来同步显示歌词。一个 LRC 歌词文件其内容格式大致如下所示：

```
[ti:匆匆那年]
[ar:王菲]
[al:电影《匆匆那年》主题曲]
[t_time:(04:08)]
[00:32.16] 匆匆那年 我们究竟说了几遍
[00:34.82] 再见之后再拖延
[00:37.62] 可惜谁没有爱过
[00:39.37] 不是一场七情上面的雄辩
[00:43.42] 匆匆那年 我们一时匆忙撂下
[00:45.82] 难以承受的诺言 只有等别人兑现
[00:54.69] 不怪那吻痕 还没积累成茧
[01:00.44] 拥抱着冬眠 也没能羽化再成仙
[01:05.74] 不怪这一段情没空反复再排练
[01:11.24] 是岁月宽容恩赐 反悔的时间
[01:22.38] 如果再见不能红着眼
[01:25.38] 是否还能红着脸
[01:28.24] 就像那年匆促刻下
[01:30.14] 永远一起那样美丽的谣言
[01:33.49] 如果过去还值得眷恋
[01:36.89] 别太快冰释前嫌
[01:39.44] 谁甘心就这样
[01:42.39] 彼此无挂也无牵
[01:45.93] 我们要互相亏欠
[01:50.94] 要不然凭何怀念
[02:02.25] 匆匆那年 我们见过太少世面
[02:04.79] 只爱看同一张脸
[02:07.65] 那么莫名其妙 那么讨人喜欢
[02:10.25] 闹起来又太讨厌
[02:13.30] 相爱那年活该匆匆
[02:15.51] 因为我们不懂顽固的诺言
[02:18.85] 只是分手的前言
[02:24.65] 不怪那天太冷 泪滴水成冰
[02:30.55] 春风也一样没吹进凝固的照片
```

[02:35.70] 不怪每一个人没能完整爱一遍

[02:41.35] 是岁月善意落下 残缺的悬念

[02:52.36] 如果再见不能红着眼 是否还能红着脸

歌词文件中 ti 标签对应歌曲的名称, ar 标签对应歌手名称, al 标签对应歌曲专辑。t_time 对应歌曲的时间, 之后的时间标签代表某一时刻对应的歌词。

使用 Xcode 创建一个名为 MyPlayer 的工程, 向其中导入一些音频文件及其对应的 LRC 歌词文件。需要注意的是, LRC 文件的文件名要与对应的歌曲名相同, 便于在解析时将歌词文件与歌曲对应。



提示

在向工程中添加大量文件时, 可以通过建立新的引用目录使工程的目录结构看起来整齐一些, 在 Xcode 文件导航区单击右键, 选择 New Group 选项即可新建一个引用目录, 如图 5-30 所示。

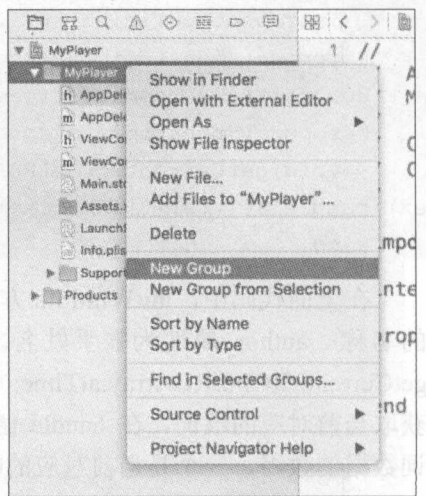


图 5-30 在工程中新建引用目录

5.5.2 LRC 歌词解析引擎的设计

在 Xcode 中创建两个引用目录 Song 和 LRC, 分别用来存放歌曲文件与歌词文件。设计一个新的类作为每行歌词的数据模型, 使用 Xcode 新建一个类文件, 命名为 LRCItem, 使其继承于 NSObject。

在 LRCItem.h 文件中添加如下方法和属性的声明。

```
@interface LRCItem : NSObject
@property (nonatomic) float time;
@property (nonatomic, copy) NSString *lrc;
//排序方法
- (BOOL)isTimeOlderThanAnother:(LRCItem *)item;
@end
```

上面声明的属性和方法中, float 是此行歌词的出现时间, lrc 是此行歌词具体的文本数据。IsTimeOlderThanAnother:方法用于行歌词数据模型的排序, 这个方法将按照时间先后进行排序。在 LRCItem.m 文件中添加方法的实现代码。

```

- (BOOL)isTimeOlderThanAnother:(LRCItem *)item{
    return self.time > item.time;
}

```

再新建一个类文件命名为 LRCEngine 作为歌词解析引擎，使其继承于 NSObject。在 LRCEngine.h 文件中引入 LRCItem 类的头文件：

```
#import "LRCItem.h"
```

在 LRCEngine.h 文件中声明如下的属性与方法：

```

@interface LRCEngine : NSObject
- (instancetype)initWithFile:(NSString *)fileName;
@property(nonatomic, strong) NSString * author;
@property(nonatomic, strong) NSString * album;
@property(nonatomic, strong) NSString * title;
- (void)getCurrentLrcInLRCArray:(void (^)(NSArray * lrcArray, int currentIndex))handle atTime:(float)time;
@end

```

在上面代码中，initWithFile:方法用于进行歌词引擎的初始化，fileName 参数为歌词文件的名称。author 属性为歌手姓名。album 属性为专辑的名称。title 属性为歌曲的名称。getCurrentLrcInLRCArray:atTime:方法为歌词引擎的核心方法，其通过传入一个时间点的值来获取当前对应的歌词，在 handle 函数块中将传入两个参数，一个是已经按时间排序的每行歌词数据的数组，一个是当前对应的歌词在数组中的位置。

在 LRCEngine.m 文件中声明一个可变数组用于存放每行歌词数据，代码如下：

```

@implementation LRCEngine
{
    NSMutableArray * _lrcArray;
}
@end

```

实现 LRCEngine 类的初始化方法，代码如下：

```

- (instancetype)initWithFile:(NSString *)fileName{
    if (self=[super init]) {
        _lrcArray = [[NSMutableArray alloc] init];
        [self creatDataWithFile:fileName];
    }
    return self;
}

```

上面方法中进行数组对象的初始化操作，creatDataWithFile:方法的实现如下：

```

- (void)creatDataWithFile:(NSString *)fileName{
    //读取文件

```



```

    NSString * lrcPath = [[NSBundle mainBundle]pathForResource:fileName ofType:@"lrc"];
    NSError * error;
    NSString * dataStr = [NSString stringWithContentsOfFile:lrcPath encoding:NSUTF8StringEncoding error:&error];
    //去掉/r
    NSMutableString * tmpStr = [[NSMutableString alloc]init];
    NSArray * tmpArray = [dataStr componentsSeparatedByString:@"\r"];
    for (int i=0; i<tmpArray.count; i++) {
        [tmpStr appendString:tmpArray[i]];
    }
    //按照换行符进行字符串分割
    NSArray * lrcArray = [tmpStr componentsSeparatedByString:@"\n"];
    //数据解析并将空数据去掉
    for (NSString * lrcStr in lrcArray) {
        if (lrcStr.length==0) {
            continue;
        }
        //判断是歌词数据还是文件信息数据
        unichar c = [lrcStr characterAtIndex:1];
        if (c>='0'&&c<='9') {
            //是歌词数据
            [self getLrcData:lrcStr];
        }else{
            //是文件信息数据
            [self getInfoData:lrcStr];
        }
    }
    //进行歌词数据的重新排序
    [_lrcArray sortedArrayUsingSelector:@selector(isTimeOlderThanAnother:)];
}

```

getLrcData:方法的实现如下:

```

-(void)getLrcData:(NSString *)lrcStr{
    //按照]进行分割
    NSArray * arr = [lrcStr componentsSeparatedByString:@"]"];
    //解析时间 同一行歌词可能对应多个时间 最后一个元素是歌词
    for (int i=0; i<arr.count-1; i++) {
        //去掉[号
        NSString *timeStr = [arr[i] substringFromIndex:1];
        //把时间字符串转换成s为单位
        NSArray * timeArr = [timeStr componentsSeparatedByString:@":"];
        float min = [timeArr[0] floatValue];
    }
}

```

```

        float sec = [timeArr[1] floatValue];
        //创建模型
        LRCItem * item = [[LRCItem alloc] init];
        item.time=min*60+sec;
        item.lrc = [arr lastObject];
        [_lrcArray addObject:item];
    }
}

```

getInfoData:方法的实现如下:

```

-(void)getInfoData:(NSString *)lrcStr{
    NSArray * arr = [lrcStr componentsSeparatedByString:@":"];
    //获取内容长度 带]符号
    NSInteger len = [arr[1] length];
    if ([arr[0] isEqualToString:@"[ti"]){
        _title = [arr[1] substringToIndex:len-1];
    }else if ([arr[0] isEqualToString:@"[ar"]){
        _author = [arr[1] substringToIndex:len-1];
    }else if ([arr[0] isEqualToString:@"[al"]){
        _alume = [arr[1] substringToIndex:len-1];
    }
}

```

实现 LRCEngine.h 中声明的 getCurrentLRCInLRCArray:atTime:方法如下:

```

-(void)getCurrentLRCInLRCArray:(void (^)(NSArray *, int))handle atTime:(float)time{
    if (!_lrcArray.count) {
        handle(nil, 0);
    }
    //找到第一个时间大于 time 的歌词位置
    int index = -2;
    for (int i=0; i<_lrcArray.count; i++) {
        float lrcTime = [_lrcArray[i] time];
        if (lrcTime>time) {
            index=i-1;
            break;
        }
    }
    if (index== -1) {
        //第一条数据
        index=0;
    }else if (index== -2){
        //没有更大的时间了 最后一条数据
    }
}

```

```

        index=(int)_lrcArray.count-1;
    }
    handle(_lrcArray,index);
}

```

为了验证 LRC 歌词引擎是否正常工作, 在 ViewController.m 文件中引入 LRCEngine 类的头文件并在 viewDidLoad 方法中编写如下代码:

```

- (void)viewDidLoad {
    [super viewDidLoad];
    LRCEngine * engine = [[LRCEngine alloc] initWithFile:@"匆匆那年"];
    [engine getCurrentLRCInLRCArray:^(NSArray *lrcArray, int currentIndex)
    {
        if (lrcArray) {
            NSLog(@"%@\\n=====\\n%@", lrcArray, [lrcArray[currentIndex] lrc
c]);
        }
    } atTime:100];
}

```

上面的代码中获取到歌词文件《匆匆那年》第 100 秒时的歌词, 打印结果如图 5-31 所示, 则说明 LRC 歌词引擎工作顺利正常。



图 5-31 LRCEngine 歌词引擎的工作打印调试

5.5.3 核心播放器引擎的设计

本节将再封装一个模块作为应用的核心播放器引擎, 这个引擎应该可以满足常规的音频播放需求, 例如循环播放、随机播放、上一曲和下一曲等。在设计之前, 先将工程设置为支持后台音频播放, 其实在 Xcode 中设置支持后台播放的方法除了前面介绍的配置 info.plist 文件外, 还可以通过另一种方式实现。

单击工程文件, 选择其中的 Capabilities 项, 在其中找到 Background Modes 一项并将其打开, 在后台运行模式中勾选支持音频后台播放的选项, 过程如图 5-32 所示。

在工程中创建一个新的类文件, 使其继承于 NSObject 类, 将其命名为 MyMusicPlayer 作为核心音频播放引擎类。在 MyMusicPlayer.h 文件中引入如下头文件:


```
#import <AVFoundation/AVFoundation.h>
```

还需要在 MyMusicPlayer.h 文件中声明一个协议，这个协议中约定当一个音频文件播放完之后的代理回调方法，提供给外界进行逻辑操作。代码如下：

```
@protocol MyMusicPlayerDelegate<NSObject>
- (void)musicPlayEndAndWillContinuePlaying: (BOOL)play;
@end
```

协议中 musicPlayEndAndWillContinuePlaying: 方法当一个音频播放完毕之后执行，其中传入的 BOOL 值参数决定是否自动播放下一个音频数据。

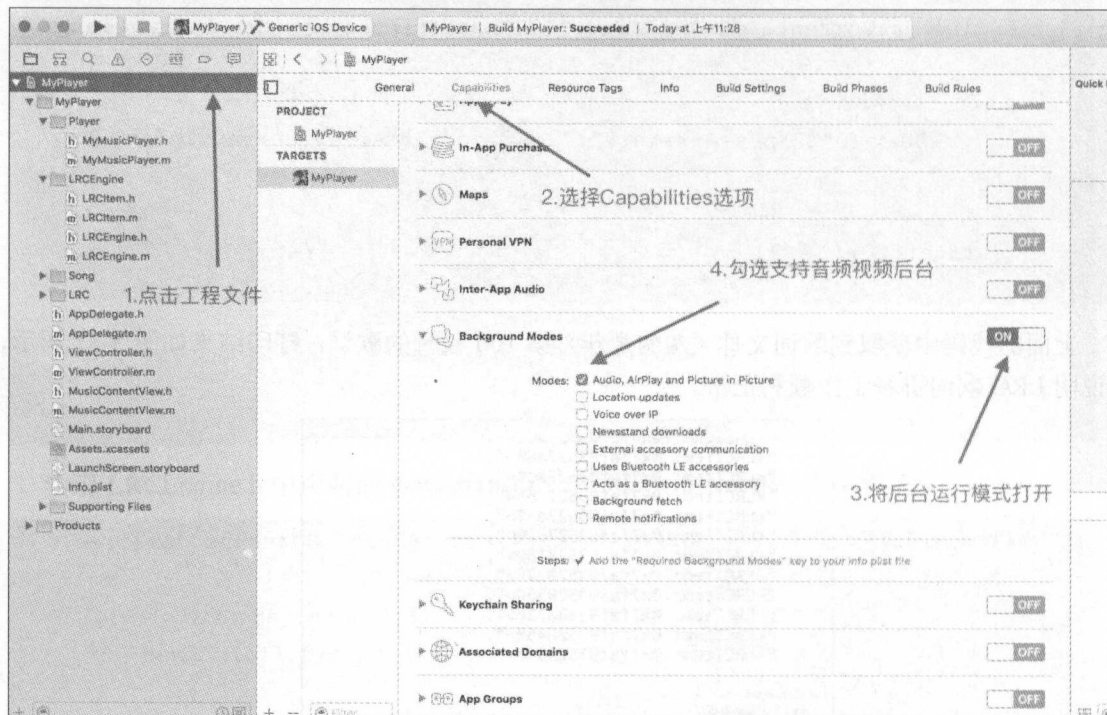


图 5-32 设置应用程序支持后台运行模式

在 MyMusicPlayer.h 文件中声明如下属性与方法：

```
@interface MyMusicPlayer : NSObject<AVAudioPlayerDelegate>
//歌曲名数组
@property(nonatomic,strong)NSArray * songsArray;
//对应歌曲的歌词名数组
@property(nonatomic,strong)NSArray * lrscArray;
//是否循环播放
@property(nonatomic,assign)BOOL isRunLoop;
//是否随机播放
@property(nonatomic,assign)BOOL isRandom;
//音频播放器是否正在播放音频
@property(nonatomic,assign)BOOL isPlaying;
```

```

//代理对象
@property(n nonatomic, weak) id<MyMusicPlayerDelegate> delegate;
//获取当前播放的是第几个音频
@property(n nonatomic, assign) int currentIndex;
//当前播放的音频文件的时长
@property(n nonatomic, assign) int currentSongTime;
//当前播放的音频文件已经播放的时长
@property(n nonatomic, assign) int hadPlayTime;
//开始播放
-(void)play;
//暂停播放
-(void)stop;
//进行继续播放与暂停播放的切换
-(void)playOrStop;
//上一曲
-(void)lastMusic;
//下一曲
-(void)nextMusic;
//停止播放
-(void)end;
//播放指定的音频文件
-(void)playAtIndex:(int)index isPlay:(BOOL)play;
@end

```

在前面的章节有介绍, 关于音频播放的后台交互与耳机线控的操作是在 `AppDelegate` 类中进行的, 因此开发者需要将 `MyMusicPlayer` 对象与程序的 `AppDelegate` 对象进行关联, 便于后台交互操作的下发到具体视图控制器中, 在 `MyMusicPlayer.m` 文件中引入如下头文件:

```
#import "AppDelegate.h"
```

在 `AppDelegate.h` 文件中导入 `MyMusicPlayer.h` 头文件并添加如下属性:

```
@property (nonatomic, strong) MyMusicPlayer *play;
```

在 `MyMusicPlayer.m` 文件中声明如下的内部属性:

```

@implementation MyMusicPlayer
{
    AVAudioPlayer * _player;
    NSTimer * _timer;
}
@end

```

在上面声明的属性中, `_player` 用于处理音频的播放, `_timer` 用于进行播放时间的更新。在 `MyMusicPlayer.m` 文件中实现类的初始化方法, 如下所示:

```

- (instancetype)init
{
    self = [super init];
    if (self) {
        _timer = [NSTimer scheduledTimerWithTimeInterval:1/60.0 target:self
selector:@selector(update) userInfo:nil repeats:YES];
        [[NSRunLoop mainRunLoop] addTimer:_timer forMode:NSRunLoopCommonModes];

        AppDelegate * delegate =[UIApplication sharedApplication].delegate;
        delegate.play=self;
    }
    return self;
}

```

在上面的初始化方法中对定时器进行创建并将当前对象与程序的 AppDelegate 对象进行了关联。

实现定时器的刷新方法 update 如下所示：

```

-(void)update{
    if (_player) {
        _hadPlayTime = _player.currentTime;
    }
}

```

实现在 MyMusicPlayer.h 文件中声明的相关方法如下所示：

```

//进行播放与暂停的切换
-(void)playOrStop{
    //先判断是否正在播放
    if (self.isPlaying) {
        //已经在播放则进行停止播放操作
        [self stop];
    }else{
        //没有在播放则进行播放操作
        [self play];
    }
}
-(void)play{
    //判断 AVAudioPlayer 对象是否存在
    if (_player!=nil) {
        [_player play];
        _isPlaying=YES;
        return;
    }else{
        //从歌曲数组中读取第一个元素

```



```

        NSString * path = [[NSBundle mainBundle]pathForResource:[self.song
sArray objectAtIndex:0] ofType:@"mp3"];
        NSURL * url = [NSURL fileURLWithPath:path];
        _player = [[AVAudioPlayer alloc] initWithContentsOfURL:url error:nil];
        _player.delegate=self;
        [_player play];
        _isPlaying=YES;
        _currentIndex=0;
        _currentSongTime=_player.duration;
    }
}

-(void)stop{
    if (_player.isPlaying) {
        [_player stop];
        _isPlaying=NO;
    }
}

-(void)end{
    [_player stop];
    _isPlaying=NO;
    _player=nil;
}

-(void)playAtIndex:(int)index isPlay:(BOOL)play{
    [_player stop];
    _isPlaying=NO;
    _player = nil;
    NSString * path = [[NSBundle mainBundle]pathForResource:[self.songsArr
ay objectAtIndex:index] ofType:@"mp3"];
    NSURL * url = [NSURL fileURLWithPath:path];
    _player = [[AVAudioPlayer alloc] initWithContentsOfURL:url error:nil];
    _player.delegate=self;
    if (play) {
        [_player play];
        _isPlaying=YES;
    }
    _currentIndex=index;
    _currentSongTime = _player.duration;
}

-(void)nextMusic{
    BOOL play = _player.isPlaying;

    [_player stop];
    _isPlaying=NO;

```

```

    _player=nil;
    //是否是最后一曲
    if (_currentIndex<self.songsArray.count-1) {
        _currentIndex++;
    }else{
        _currentIndex=0;
    }
    //是否随机播放
    if (self.isRandom) {
        unsigned long max = self.songsArray.count;
        _currentIndex = arc4random()%max;
    }
    NSString * path = [[NSBundle mainBundle]pathForResource:[self.songsArray objectAtIndex:_currentIndex] ofType:@"mp3"];
    NSURL * url = [NSURL fileURLWithPath:path];
    _player = [[AVAudioPlayer alloc] initWithContentsOfURL:url error:nil];
    _currentSongTime=_player.duration;
    _player.delegate=self;
    if (play) {
        [_player play];
        _isPlaying=YES;
    }

}

-(void)lastMusic{
    BOOL play = _player.isPlaying;
    [_player stop];
    _isPlaying=NO;
    _player=nil;
    if (_currentIndex>0) {
        _currentIndex--;
    }else{
        _currentIndex=(int)_songsArray.count-1;
    }
    if (self.isRandom) {
        unsigned long max = self.songsArray.count;
        _currentIndex = arc4random()%max;
    }
    NSString * path = [[NSBundle mainBundle]pathForResource:[self.songsArray objectAtIndex:_currentIndex] ofType:@"mp3"];
    NSURL * url = [NSURL fileURLWithPath:path];
    _player = [[AVAudioPlayer alloc] initWithContentsOfURL:url error:nil];
    _currentSongTime=_player.duration;

```



```

    _player.delegate=self;
    if (play) {
        [_player play];
        _isPlaying=YES;
    }
}

```

在 MyMusicPlayer.m 文件中还需要实现一个 AVAudioPlayerDelegate 协议中约定的方法: audioPlayerDidFinishPlaying:successfully:方法,这个方法在 AVAudioPlayer 播放结束后会被调用,实现方法如下:

```

- (void)audioPlayerDidFinishPlaying:(AVAudioPlayer *)player successfully:
(BOOL)flag{
    _player = nil;
    _isPlaying=NO;
    //是否循环播放
    if (_isRandom) {
        unsigned long max = self.songsArray.count;
        int songIndex = arc4random()%max;
        NSString * path = [[NSBundle mainBundle]pathForResource:[self.song
sArray objectAtIndex:songIndex] ofType:@"mp3"];
        NSURL * url = [NSURL fileURLWithPath:path];
        _player = [[AVAudioPlayer alloc] initWithContentsOfURL:url error:nil];
        _player.delegate=self;
        [_player play];
        _isPlaying=YES;
        [self.delegate musicPlayEndAndWillContinuePlaying:YES];
        _currentIndex=songIndex;
        _currentSongTime=_player.duration;
        return;
    }
    if (_currentIndex<self.songsArray.count-1) {
        //是否是最后一首
        NSString * path = [[NSBundle mainBundle]pathForResource:[self.song
sArray objectAtIndex:++_currentIndex] ofType:@"mp3"];
        NSURL * url = [NSURL fileURLWithPath:path];
        _player = [[AVAudioPlayer alloc] initWithContentsOfURL:url error:nil];
        _currentSongTime=_player.duration;
        _player.delegate=self;
        [_player play];
        _isPlaying=YES;
        [self.delegate musicPlayEndAndWillContinuePlaying:YES];
    }else if (_currentIndex==self.songsArray.count-1){
        //是否循环

```



```

        if (!_isRunLoop) {
            _currentIndex=0;
            NSString * path = [[NSBundle mainBundle]pathForResource:[self.songsArray objectAtIndex:_currentIndex] ofType:@"mp3"];
            NSURL * url = [NSURL fileURLWithPath:path];
            _player = [[AVAudioPlayer alloc] initWithContentsOfURL:url error:nil];

            _player.delegate=self;
            _currentSongTime=_player.duration;
            [_player play];
            _isPlaying=YES;
            [self.delegate musicPlayEndAndWillContinuePlaying:YES];
        }else{
            [self.delegate musicPlayEndAndWillContinuePlaying:NO];
        }
    }
}

```

5.5.4 歌曲列表与歌词显示视图界面的设计

前面小节中设计的两个模块：歌词引擎模块和播放引擎模块都属于工具类模块，本节将设计一个视图类模块，歌曲列表与歌词显示视图应支持左右滑动进行界面模式的切换，左边模式为歌曲列表与单行歌词显示控件，右边模式为多行歌词显示控件。

在工程中创建一个新的类文件，取名为 `MusicContentView`，使其继承于 `UIView` 类。因为 `MusicContentView` 中包含歌曲列表，单击歌曲列表中的歌曲应该支持播放歌曲的切换。因此 `MusicContentView` 中应该能够操作前面设计的音频播放引擎对象，在 `MusicContentView.h` 文件中引入如下头文件并声明如下属性和方法：

```

#import "MyMusicPlayer.h"

@interface MusicContentView : UIView
//歌曲列表数据源数组
@property(nonatomic,strong)NSArray * titleDataAttay;
//这个方法设置当前界面显示的歌词 对应歌曲播放的相应时间
-(void)setCurretLRCArray:(NSArray *)array index:(int)index;
//播放器引擎对象的引用
@property(nonatomic,strong)MyMusicPlayer * play;
//锁屏界面要显示的图片
@property(nonatomic,readonly)UIImage * lrcImage;
@end

```

前面小节中介绍过，iOS 锁屏界面只支持显示图片，若想在其中显示滚动的歌词，开发者需要不停的刷新锁屏界面的图片，上面的 `lrcImage` 属性为播放歌曲当前时刻对应的歌词图片对象。

在 MusicContentView.m 文件中引入如下头文件：

```
#import "LRCItem.h"
```

歌曲列表可以采用 UITableView 来设计，在 MusicContentView.m 文件中添加遵守相应协议的代码。

```
@interface MusicContentView() <UITableViewDataSource, UITableViewDelegate>
@end
```

在 MusicContentView.m 文件中声明如下属性：

```
@implementation MusicContentView
{
    UIScrollView * _scrollView;
    //歌曲列表视图
    UITableView * _titleTableView;
    //单行显示的歌词显示标签
    UILabel * _lrcLabel;
    //锁屏图片中的歌词标签
    UILabel * _lrcIMGLabel;
    //锁屏图片的背景
    UIImageView * _lrcIMGbg;
    //多行显示的歌词显示标签
    UILabel * _lrcView;
    //多行显示歌词视图的显示行数
    int _lines;
}
@end
```

在 MusicContentView.m 文件中实现类的初始化方法，如下所示：

```
- (instancetype) initWithFrame:(CGRect) frame
{
    self = [super initWithFrame:frame];
    if (self) {
        //设置视图背景为透明色
        self.backgroundColor = [UIColor clearColor];
        //初始化滚动视图
        _scrollView = [[UIScrollView alloc] initWithFrame:CGRectMake(0, 0,
frame.size.width, frame.size.height)];
        [self addSubview:_scrollView];
        _scrollView.backgroundColor = [UIColor clearColor];
        //初始化歌曲列表
        _titleTableView = [[UITableView alloc] initWithFrame:CGRectMake(40,
0, frame.size.width-90, frame.size.height-40) style:UITableViewStylePlain];
```

```

        _titleTableView.backgroundColor = [UIColor clearColor];
        _titleTableView.delegate=self;
        _titleTableView.dataSource=self;
        //设置表格视图行间无分割线
        _titleTableView.separatorStyle = UITableViewCellStyleNone;
        [_scrollView addSubview:_titleTableView];
        //设置滚地视图的可滚动范围
        _scrollView.contentSize = CGSizeMake(frame.size.width*2, frame.size.height);
        _scrollView.showsHorizontalScrollIndicator=NO;
        //设置滚动视图翻页效果
        _scrollView.pagingEnabled=YES;
        //初始化单行显示的歌词控件
        _lrcLabel = [[UILabel alloc] initWithFrame:CGRectMake(20, frame.size.height-50, frame.size.width-40, 50)];
        _lrcLabel.backgroundColor = [UIColor clearColor];
        //设置歌词颜色为白色
        _lrcLabel.textColor = [UIColor whiteColor];
        [_scrollView addSubview:_lrcLabel];
        _lrcLabel.textAlignment = NSTextAlignmentCenter;
        _lrcLabel.numberOfLines=0;
        //初始化多行显示的歌词控件
        _lrcView = [[UILabel alloc] initWithFrame:CGRectMake(frame.size.width+20, 50, frame.size.width-40, frame.size.height-100)];
        //根据屏幕尺寸获取显示行数
        _lines = (int)_lrcView.frame.size.height/21;
        _lrcView.numberOfLines = _lines;
        _lrcView.textAlignment = NSTextAlignmentCenter;
        _lrcView.textColor = [UIColor whiteColor];
        [_scrollView addSubview:_lrcView];
        //初始化锁屏图片上的歌词标签
        _lrcIMGLabel = [[UILabel alloc] initWithFrame:CGRectMake(20, 0, self.frame.size.width-40, self.frame.size.height)];
        _lrcIMGLabel.numberOfLines = _lines;
        _lrcIMGLabel.textAlignment = NSTextAlignmentCenter;
        _lrcIMGLabel.textColor = [UIColor whiteColor];

    }

    return self;
}

```

在 MusicContentView.m 中重写实现 titledataArray 数据源的 setter 方法，如下所示：


```

-(void)setTitleDataAttay:(NSArray *)titleDataAttay{
    _titleDataAttay = [NSArray arrayWithArray:titleDataAttay];
    [_titleTableView reloadData];
}

```

在 MusicContentView.m 中实现 UITableView 控件的代理与数据源协议中的相应方法如下所示:

```

-(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView{
    //设置分区数为1
    return 1;
}

-(NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section{
    //设置行数为数据源中数据个数
    return self.titleDataAttay.count;
}

-(UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath{
    UITableViewCell * cell = [tableView dequeueReusableCellWithIdentifier:@"cellId"];
    if (cell==nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"cellId"];
        cell.backgroundColor = [UIColor clearColor];
        cell.textLabel.textColor = [UIColor whiteColor];
        //设置 cell 的选中效果为无
        cell.selectionStyle = UITableViewCellSelectionStyleNone;
    }
    cell.textLabel.text = self.titleDataAttay[indexPath.row];
    return cell;
}

-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath{
    //单击歌曲列表中某行后播放相应的歌曲
    [self.play playAtIndex:(int)indexPath.row isPlay:self.play.isPlaying];
}

```

在 MusicContentView.m 中实现 setCurrentLRCArray:index:方法如下所示:

```

-(void)setCurretLRCArray:(NSArray *)array index:(int)index{
    NSString * lineLRC = [(LRCItem *)array[index] lrc];
    _lrcLabel.text = lineLRC;
    //进行行数设置
}

```

```

NSMutableString * lrcStr = [[NSMutableString alloc] init];
if (index < _lines/2) {
    //前面用\n 补齐
    int offset = (int)_lines/2-index;
    for (int j=0; j<offset; j++) {
        [lrcStr appendFormat:@"%n"];
    }
    for (int j=0; j<_lines-offset; j++) {
        [lrcStr appendFormat:@"%s\n", [(LRCItem *)array[j] lrc]];
    }
} else if (array.count-1-index < _lines/2) {
    //后面用\n 补齐
    int offset = (int)_lines/2-(int)(array.count-index-1);
    for (int j=index-(int)_lines/2; j<array.count; j++) {
        [lrcStr appendFormat:@"%s\n", [(LRCItem *)array[j] lrc]];
    }
    for (int j=0; j<offset; j++) {
        [lrcStr appendFormat:@"%n"];
    }
} else {
    for (int j=0; j<_lines; j++) {
        [lrcStr appendString:[(LRCItem *)array[index-_lines/2+j] lrc]];

        [lrcStr appendString:@"%n"];
    }
}

NSMutableAttributedString * attriStr = [[NSMutableAttributedString alloc] initWithString:lrcStr];
NSRange range = [lrcStr rangeOfString:[array[index] lrc]];
[attriStr setAttributes:@{NSForegroundColorAttributeName:[UIColor greenColor]} range:range];
_lrcView.attributedText = attriStr;
_lrcIMGLabel.attributedText = attriStr;
//进行截屏
if (!_lrcIMGbg) {
    _lrcIMGbg = [[UIImageView alloc] initWithFrame:CGRectMake(0, 0, self.frame.size.width, self.frame.size.height)];
    _lrcIMGbg.image = [UIImage imageNamed:@"BG.jpeg"];
    [_lrcIMGbg addSubview:_lrcIMGLabel];
}
UIGraphicsBeginImageContext(_lrcIMGbg.frame.size);
CGContextRef context = UIGraphicsGetCurrentContext();
[_lrcIMGbg.layer renderInContext:context];

```



```

UIImage *img = UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();
_lrcImage = [img copy];
}

```

5.5.5 播放器主页面的实现

前面小节中所做的工具类模块与视图类模块都是独立的完成某些功能或者显示某种视图，本小节将在 `ViewController` 类中将其进行组合与协调，完成音频播放器的开发。

在 `ViewController.m` 文件中引入如下头文件：

```

#import "LRCEngine.h"
#import "MyMusicPlayer.h"
#import "MusicContentView.h"
#import <MediaPlayer/MediaPlayer.h>

```

在 `ViewController.m` 中编写遵守相关协议的代码并声明一些属性如下所示：

```

@interface ViewController ()<MyMusicPlayerDelegate>
{
    MyMusicPlayer * _player;
    //内容视图
    MusicContentView * _contentView;
    //标题标签
    UILabel * _titleLabel;
    //进度条
    UIProgressView * _progress;
    //播放按钮
    UIButton * _playBtn;
    //下一曲按钮
    UIButton * _nextBtn;
    //上一曲按钮
    UIButton * _lastBtn;
    //循环播放按钮
    UIButton * _circleBtn;
    //随机播放按钮
    UIButton * _randomBtn;
    //存放歌曲名
    NSArray * _dataArray;
    NSTimer * _timer;
}
@end

```

在 `ViewController.m` 文件的 `viewDidLoad` 方法中调用如下方法：


```

- (void)viewDidLoad {
    [super viewDidLoad];
    //创建数据
    [self creatData];
    //创建播放模块
    [self creatPlayer];
    //创建视图模块
    [self creatView];
    //进行刷新 UI 操作
    [self updateUI];
}

```

creatData 方法的实现如下所示：

```

- (void)creatData{
    _dataArray = @[@"匆匆那年",@"致青春",@"清风徐来",@"矜持",@"暗涌",@"天空",@"
    容易受伤的女人",@"清平调",@"但愿人长久",@"暧昧",@"执迷不悔",@"约定",@"我愿意",@"棋子",
    @"梦醒了",@"影子",@"人间",@"爱与痛的边缘",@"旋木",@"红豆",@"传奇",@"爱不可及"];
}

```

creatData 方法对数据源进行了创建，前提是要将上面数组中歌名对应的音频文件和歌词文件都导入项目中，歌词文件要与音频文件名称对应，为了使工程结构看起来更整洁，开发者可以将歌曲与歌词文件分别放于相应的目录下，如图 5-33 所示。

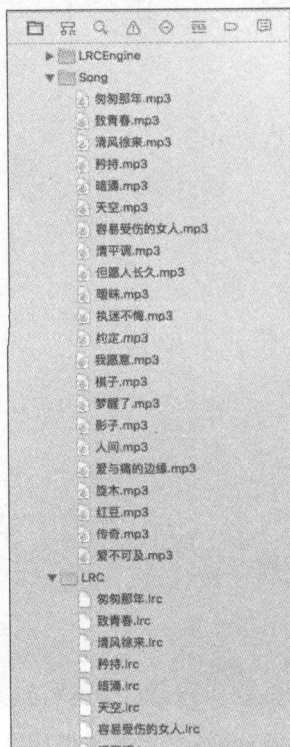


图 5-33 Xcode 的工程目录结构

creatPlayer 方法的实现如下:

```
-(void)creatPlayer{
    _player = [[MyMusicPlayer alloc]init];
    _player.songsArray=_dataArray;
    NSMutableArray * mulArr = [[NSMutableArray alloc]init];
    for (int i=0; i<_dataArray.count; i++) {
        //进行歌词模块创建
        LRCEngine * engine = [[LRCEngine alloc]initWithFile:_dataArray
[i]];

        [mulArr addObject:engine];
    }
    _player.lrcsArray = mulArr;
    _player.delegate=self;
}
```

createView 方法的实现如下:

```
-(void)createView{
    //创建背景
    UIImageView * bg = [[UIImageView alloc]initWithFrame:self.view.bounds];
    bg.image = [UIImage imageNamed:@"BG.jpeg"];
    //设置为可接收用户交互
    bg.userInteractionEnabled=YES;
    [self.view addSubview:bg];
    //创建歌曲标题 Label
    _titleLabel = [[UILabel alloc]initWithFrame:CGRectMake(0, 20, bg.frame.
size.width, 40)];
    _titleLabel.font = [UIFont boldSystemFontOfSize:22];
    _titleLabel.textAlignment = NSTextAlignmentCenter;
    _titleLabel.text = _dataArray[0];
    _titleLabel.backgroundColor = [UIColor clearColor];
    _titleLabel.textColor = [UIColor whiteColor];
    [bg addSubview:_titleLabel];
    //创建歌曲进度条
    _progress = [[UIProgressView alloc]initWithProgressViewStyle:UIProgre
ssViewStyleDefault];
    _progress.progressTintColor=[UIColor redColor];
    _progress.trackTintColor = [UIColor whiteColor];
    _progress.frame=CGRectMake(20, self.view.frame.size.height-70, self.v
iew.frame.size.width-40, 5);
    [bg addSubview:_progress];
    //创建播放按钮
    _playBtn = [UIButton buttonWithType:UIButtonTypeCustom];
```

```

        [_playBtn setBackgroundImage:[UIImage imageNamed:@"play"] forState:UIControlStateNormal];
        [_playBtn setBackgroundImage:[UIImage imageNamed:@"pause"] forState:UIControlStateSelected];
        _playBtn.frame=CGRectMake(self.view.frame.size.width/2-20, self.view.frame.size.height-45, 40, 30);
        [_playBtn addTarget:self action:@selector(playMusic) forControlEvents:UIControlEventTouchUpInside];
        [bg addSubview:_playBtn];
        //创建下一曲按钮
        _nextBtn = [UIButton buttonWithType:UIButtonTypeCustom];
        _nextBtn.frame=CGRectMake(self.view.frame.size.width/2+40, self.view.frame.size.height-45, 40, 30);
        [_nextBtn setBackgroundImage:[UIImage imageNamed:@"nextMusic"] forState:UIControlStateNormal];
        [_nextBtn addTarget:self action:@selector(next) forControlEvents:UIControlEventTouchUpInside];
        [bg addSubview:_nextBtn];
        //创建上一曲按钮
        _lastBtn = [UIButton buttonWithType:UIButtonTypeCustom];
        _lastBtn.frame = CGRectMake(self.view.frame.size.width/2-80, self.view.frame.size.height-45, 40, 30);
        [_lastBtn setBackgroundImage:[UIImage imageNamed:@"aboveMusic"] forState:UIControlStateNormal];
        [_lastBtn addTarget:self action:@selector(last) forControlEvents:UIControlEventTouchUpInside];
        [bg addSubview:_lastBtn];
        //创建循环播放按钮
        _circleBtn = [UIButton buttonWithType:UIButtonTypeCustom];
        _circleBtn.frame = CGRectMake(self.view.frame.size.width/2-140, self.view.frame.size.height-45, 40, 30);
        [_circleBtn setBackgroundImage:[UIImage imageNamed:@"circleClose"] forState:UIControlStateNormal];
        [_circleBtn setBackgroundImage:[UIImage imageNamed:@"circleOpen"] forState:UIControlStateSelected];
        [_circleBtn addTarget:self action:@selector(circle) forControlEvents:UIControlEventTouchUpInside];
        [bg addSubview:_circleBtn];
        //创建随机播放按钮
        _randomBtn = [UIButton buttonWithType:UIButtonTypeCustom];
        _randomBtn.frame=CGRectMake(self.view.frame.size.width/2+100, self.view.frame.size.height-45, 40, 30);

```



```

        [_randomBtn setBackgroundImage:[UIImage imageNamed:@"randomClose"] for
rState:UIControlStateNormal];
        [_randomBtn setBackgroundImage:[UIImage imageNamed:@"randomOpen"] for
State:UIControlStateSelected];
        [_randomBtn addTarget:self action:@selector(random) forControlEvents:
UIControlEventTouchUpInside];
        [bg addSubview:_randomBtn];
        //创建歌曲列表与歌词显示控件视图
        _contentView = [[MusicContentView alloc] initWithFrame:CGRectMake(0, 70,
self.view.frame.size.width, self.view.frame.size.height-150)];
        _contentView.titleDataAttay = _dataArray;
        _contentView.play=_player;
        [bg addSubview:_contentView];
    }

```

各个功能按钮的触发方法的实现如下:

```

-(void)playMusic{
    if (_player.isPlaying) {
        _playBtn.selected=NO;
        [_player stop];
    }else{
        _playBtn.selected=YES;
        [_player play];
    }
}

-(void)next{
    [_player nextMusic];
}

-(void)last{
    [_player lastMusic];
}

-(void)circle{
    if (_player.isRunLoop) {
        _player.isRunLoop=NO;
        _circleBtn.selected=NO;
    }else{
        _player.isRunLoop=YES;
        _circleBtn.selected=YES;
    }
}

-(void)random{
    if (_player.isRandom) {
        _player.isRandom=NO;
    }
}

```

```

        _randomBtn.selected=NO;
    }else{
        _player.isRandom=YES;
        _randomBtn.selected=YES;
    }
}

```

updateUI 方法的实现如下：

```

-(void)updateUI{
    _timer = [NSTimer scheduledTimerWithTimeInterval:1/60.0 target:self selector:@selector(update) userInfo:nil repeats:YES];
    [[NSRunLoop mainRunLoop] addTimer:_timer forMode:NSRunLoopCommonModes];
}

```

定时器的触发方法 update 的实现如下：

```

-(void)update{
    _titleLabel.text = _dataArray[_player.currentIndex];
    //更新进度条
    if (_player.hadPlayTime!=0) {
        float progress = (float)_player.hadPlayTime/_player.currentSongTime;
        _progress.progress = progress;
    }
    //更新歌词
    LRCEngine * engine = _player.lrcsArray[_player.currentIndex];
    [engine getCurrentLRCInLRCArray:^(NSArray *lrcArray, int currentIndex) {
        [_contentView setCurrentLRCArray:lrcArray index:currentIndex];
    } atTime:_player.hadPlayTime];
    //更新锁屏界面
    NSMutableDictionary *dict = [[NSMutableDictionary alloc] init];

    [dict setObject:_dataArray[_player.currentIndex] forKey:MPMediaItemPropertyTitle];
    [dict setObject:@"王菲" forKey:MPMediaItemPropertyArtist];
    [dict setObject:@"致敬天后" forKey:MPMediaItemPropertyAlbumTitle];

    UIImage *newImage = _contentView.lrcImage;
    [dict setObject:[[MPMediaItemArtwork alloc] initWithImage:newImage]
        forKey:MPMediaItemPropertyArtwork];
    [dict setObject:[NSNumber numberWithInt:_player.currentSongTime] forKey:MPMediaItemPropertyPlaybackDuration];
    [dict setObject:[NSNumber numberWithInt:_player.hadPlayTime] forKey:MPNowPlayingInfoPropertyElapsedPlaybackTime]; //音乐当前已经过时间
}

```



```
[MPNowPlayingInfoCenter defaultCenter] setNowPlayingInfo:dict];
}
```

在 `ViewController.m` 中还需要实现一首歌曲播放完毕后调用的协议方法，如下所示：

```
-(void)musicPlayEndAndWillContinuePlaying:(BOOL)play{
    if (play) {
        _playBtn.selected=YES;
    }else{
        _playBtn.selected=NO;
    }
}
```

5.5.6 后台播放音频用户交互的处理

后台播放音频的用户交互是通过系统与 `AppDelegate` 类对象实现的，在 `AppDelegate.m` 文件的 `application:didFinishLaunchingWithOptions:` 方法中添加如下代码：

```
-(BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions {
    AVAudioSession *session = [AVAudioSession sharedInstance];
    [session setActive:YES error:nil];
    [session setCategory:AVAudioSessionCategoryPlayback error:nil];
    [[UIApplication sharedApplication] beginReceivingRemoteControlEvents];
    return YES;
}
```

在 `AppDelegate.m` 中实现接收交互通知的方法如下：

```
//后台播放控制
-(void)remoteControlReceivedWithEvent:(UIEvent *)event{
    if (event.type==UIEventTypeRemoteControl) {
        switch (event.subtype) {
            case UIEventSubtypeRemoteControlPlay:
                [self.play play];
                break;
            case UIEventSubtypeRemoteControlNextTrack:
                [self.play nextMusic];
                break;
            case UIEventSubtypeRemoteControlPreviousTrack:
                [self.play lastMusic];
                break;
            case UIEventSubtypeRemoteControlPause:
                [self.play stop];
                break;
            case UIEventSubtypeRemoteControlTogglePlayPause:
```



```

        [self.play playOrStop];
        break;
    default:
        break;
    }
}
}

```

至此，《天后王菲》音频播放器应用就已经开发完成了，其中主要界面如图 5-34~图 5-37 所示。



图 5-34 歌曲列表界面

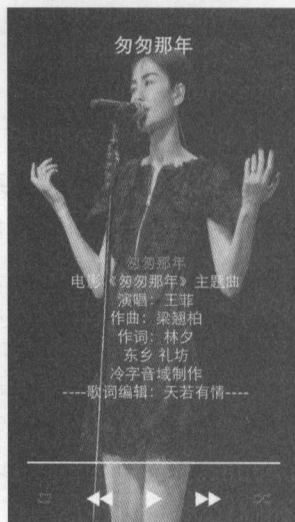


图 5-35 多行歌词显示界面



图 5-36 后台播放上拉抽屉界面



图 5-37 后台播放锁屏界面

学习之余，读者可以使用这款小应用听听音乐，放松一下，本章最后，向歌坛天后王菲致以敬意

第 6 章

动画开发

动画效果是一款应用程序的重要组成部分，应用的界面是否美观大方，交互是否让用户感到舒适友好，动画编程技术在许多类似这样人性化的需求中作用十分关键。iOS 系统的一大亮点就是它强大流畅的页面动画处理技术，在 iOS 开发中，系统提供了许多简洁易用的动画框架帮助开发者进行动画编程。本章将向读者介绍在 iOS 开发中常用的动画编程方法，力求让读者全面理解 iOS 动画开发的思路和相关框架并可以将动画编程技术应用于实战开发中。

通过本章的学习，读者能够掌握：

1. 使用 UIImageView 播放图片组动画。
2. 使用 block 回调的方式创建 UIView 层过渡动画。
3. 使用 block 回调的方式创建 UIView 层转场动画。
4. 使用 commit 方式创建 UIView 层过渡动画。
5. 使用 commit 方式创建 UIView 层转场动画。
6. CALayer 层在 UI 开发中的应用。
7. 各种 CALayer 子类的用法。
8. CoreAnimation 核心动画框架的使用。
9. 视图的 transform 变换。
10. 粒子效果动画。
11. iOS 中播放 GIF 动态图的方法。
12. 实战 Flappy Bird 游戏。

6.1 使用 UIImageView 播放图片组帧动画

UIImageView 控件对于读者来说并不陌生，在前面的章节中使用其进行了图片的展示。其实，UIImageView 除了可以进行图片的展示外，还可以对一组图片进行快速切换来形成动画效果。

使用 Xcode 创建一个名为 UIImageViewAnimationTest 的工程，向其中加入 3 张画面连续的动画图片，如图 6-1 所示。



图 6-1 3 张动作连续的小鸟飞行图片

在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIImageView * imageView = [[UIImageView alloc] initWithFrame:CGRectMake(
    100, 100, 42, 28)];
    NSMutableArray * imageArray = [[NSMutableArray alloc] init];
    for (int i=0; i<3; i++) {
        UIImage * image = [UIImage imageNamed:[NSString stringWithFormat:@"%s",
        bird%d", i+1]];
        [imageArray addObject:image];
    }
    //设置动画图片数组
    imageView.animationImages = imageArray;
    imageView.animationDuration = 1;
    imageView.animationRepeatCount = 0;
    [self.view addSubview:imageView];
    [imageView startAnimating];
}
```

上面代码中，UIImageView 对象的 animationImages 属性用于设置要播放的动画图片数组，与之对应的 UIImageView 类中还有一个 highlightedAnimationImages 属性用于设置处于高亮状态的 UIImageView 的动画图片数组。UIImageView 对象的 animationDuration 属性用于设置动画播放一遍的时长，即将动画数组中所有图片展示一遍所消耗的时间。UIImageView 对象的 animationRepeatCount 属性设置动画播放的循环次数，如果设置为 0 则为无限循环。

将 UIImageView 对象的动画数组及动画播放的相关属性设置完成后，需要调用 startAnimating 方法开始播放，与之对应，stopAnimating 方法用于停止 UIImageView 动画的播放。运行工程，效果如图 6-2 所示，小鸟在不停地扇动翅膀。

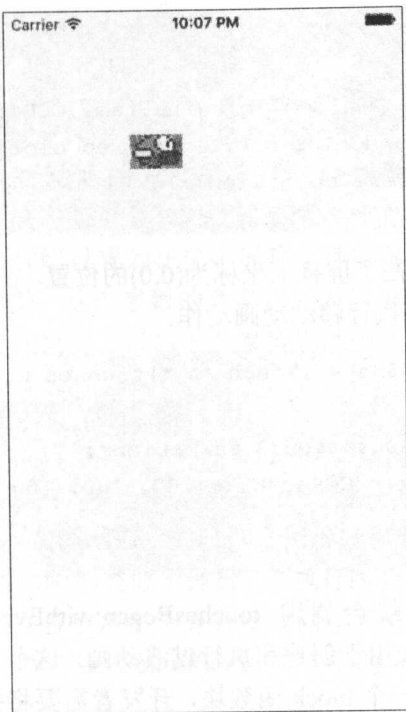


图 6-2 播放 UIImageView 图片组动画

6.2 UIView 层动画的应用

在 iOS 动画开发中，UIView 层的动画应该是开发中使用率最高，调用方法最为简单的动画开发方式了。UIView 层动画可以将视图控件一些属性的变化以动画的形式展现出来，比如视图控件背景色的渐变效果、视图控件位置改变的移动动画效果和视图控件尺寸改变的缩放动画效果等。

6.2.1 执行 UIView 层过渡动画的 3 个类方法

UIView 层中的过渡动画主要由 3 个类方法来完成。首先使用 Xcode 创建一个工程，命名为 UIViewAnimationTest。在 ViewController.m 文件中声明一个 UIView 类型的色块，作为测试动画的载体，如下所示：

```
@interface ViewController ()
{
    UIView * _colorView;
}
@end
```

在 viewDidLoad 方法中进行色块的初始化设置。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    _colorView = [[UIView alloc] initWithFrame:CGRectMake(0, 0, 100, 100)];
    _colorView.backgroundColor = [UIColor redColor];
    [self.view addSubview:_colorView];
}

```

上面代码将色块初始化放在了屏幕上坐标为(0,0)的位置。在 `ViewController.m` 文件中重写 `touchesBegan:withEvent:` 方法来执行移动动画动作。

```

- (void)touchesBegan: (NSSet<UITouch *> *)touches withEvent: (UIEvent *)event {
    [UIView animateWithDuration:3 animations:^(
        _colorView.frame = CGRectMake(100, 100, 100, 100);
    )];
}

```

当用户单击屏幕时，系统会调用 `touchesBegan:withEvent:` 方法，`UIView` 的类方法 `animateWithDuration:animations:` 用于创建和执行过渡动画，这个方法中第 1 个参数负责设置动画执行的时间，第 2 个参数是一个 block 函数块，开发者需要将执行动画的过渡操作代码放入其中，例如上面代码将色块的位置修改为(100,100)，这样当运行工程后单击屏幕，可以看到在 3 秒内，色块从(0,0)点移动到了(100,100)点。

如下方法也可以进行 `UIView` 层过渡动画的创建与执行。

```

- (void)touchesBegan: (NSSet<UITouch *> *)touches withEvent: (UIEvent *)event {
    [UIView animateWithDuration:3 animations:^(
        _colorView.frame = CGRectMake(100, 100, 100, 100);
    ) completion:^(BOOL finished) {
        NSLog(@"动画完成");
    }];
}

```

`animateWithDuration:animations:completion:` 方法比第 1 个方法后面多了 1 个参数，与第 1 个方法类似，这个方法中第 1 个参数负责设置动画执行的时间，第 2 个参数是 block 代码块，里面可以添加要执行动画效果的过渡代码，第 3 个参数也是一个 block 代码块，这个 block 中的代码将在动画执行完成之后调用，例如上面代码中打印了 Log 信息，“完成动画”这句话将在动画执行结束之后被打印。

除了上面介绍的两个创建过渡动画的方法，`UIView` 类还提供了一个更加复杂的方法，代码示例如下：

```

- (void)touchesBegan: (NSSet<UITouch *> *)touches withEvent: (UIEvent *)event {
    [UIView animateWithDuration:3 delay:1 options:UIViewAnimationOptionCurveEaseInOut animations:^(
        _colorView.frame = CGRectMake(100, 100, 100, 100);
    ) completion:^(BOOL finished) {

```

```

        NSLog(@"动画完成");
    }
}

```

上面代码中使用的创建执行动画的方法又多了一些参数，`animateWithDuration:delay:options:animations:completion:`方法中第1个参数设置动画执行的时间；第2个参数设置动画延时多长时间开始执行，例如上面代码设置为在单击屏幕1秒后再开始执行移动色块的动画；第3个参数设置动画执行的配置参数，这个参数的意义在后面的小节中会进行详细介绍；第4个参数为要执行动画动作的 block 代码块。最后一个参数为动画效果完成之后执行的代码块。

6.2.2 创建 UIView 层的阻尼动画

在各种各样的 iOS 动画效果中，有一种动画接近生活并且十分自然，在程序开发中应用甚广。例如原生的 `UITableView` 在滑动停止的时候会出现减速效果，`UITableView` 的 `bounce` 属性可以支持回弹效果等。这些动画都类似于弹簧，在执行时会有模拟的缓冲或阻尼参数。使用 `UIView` 层动画开发技术，开发者也可以十分方便地创建阻尼动画。

在 `UIViewAnimationTest` 工程 `ViewController.m` 文件中的 `touchesBegan:withEvent:`方法中添加如下代码：

```

-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{
    [UIView animateWithDuration:3 delay:1 usingSpringWithDamping:1 initialSpringVelocity:20 options:UIViewAnimationOptionCurveEaseIn animations:^(
        _colorView.frame = CGRectMake(100, 100, 100, 100);
    ) completion:nil];
}

```

上面代码使用 `animateWithDuration:delay:usingSpringWithDamping:initialSpringVelocity:options:animations:completion:`方法进行阻尼动画的创建，在这个方法中第1个参数设置动画执行的时长；第2个参数设置阻尼动画的阻尼度，这个参数的取值为0~1，越接近1则阻尼度越大。第3个参数设置动画的初速度；第4个参数设置动画的配置参数；第5个参数为要执行动画的属性改变 block 代码块；最后一个参数为动画执行完成后的回调 block 代码块。

6.2.3 动画参数配置与组合动画

在上一小节中介绍了 `animateWithDuration:delay:options:animations:completion:`方法，这个方法中第3个参数用来设置动画的配置参数，这个参数是一个 `UIViewAnimationOptions` 类型的枚举，其中定义了许多 `UIView` 层动画执行时的基础属性、动画执行时的时间函数和转场动画效果相关的枚举值。`UIViewAnimationOptions` 枚举值及其意义如下所示。

```

typedef NS_OPTIONS(NSUInteger, UIViewAnimationOptions) {
    //这部分是基础属性的设置
    UIViewAnimationOptionLayoutSubviews = 1 << 0, //设置子视图随父视图展示动画
}

```



```

    UIViewAnimationOptionAllowUserInteraction    = 1 << 1, //允许在动画执行
    时用户与其进行交互
    UIViewAnimationOptionBeginFromCurrentState    = 1 << 2, //允许在动画执行
    时执行新的动画
    UIViewAnimationOptionRepeat                    = 1 << 3, //设置动画循环执行
    UIViewAnimationOptionAutoreverse                = 1 << 4, //设置动画反向执行,
    必须和重复执行一起使用
    UIViewAnimationOptionOverrideInheritedDuration = 1 << 5, //强制动画使用内
    层动画的时间值
    UIViewAnimationOptionOverrideInheritedCurve    = 1 << 6, //强制动画使用内
    层动画曲线值
    UIViewAnimationOptionAllowAnimatedContent      = 1 << 7, //设置动画视图实
    时刷新
    UIViewAnimationOptionShowHideTransitionViews   = 1 << 8, //设置视图切换时
    隐藏, 而不是移除
    UIViewAnimationOptionOverrideInheritedOptions = 1 << 9, //
    //这部分属性设置动画播放的时间函数效果
    UIViewAnimationOptionCurveEaseInOut            = 0 << 16, //淡入淡出 首末减速
    UIViewAnimationOptionCurveEaseIn               = 1 << 16, //淡入 初始减速
    UIViewAnimationOptionCurveEaseOut              = 2 << 16, //淡出 末尾减速
    UIViewAnimationOptionCurveLinear                = 3 << 16, //线性 匀速执行
    //这部分设置 UIView 切换效果
    UIViewAnimationOptionTransitionNone             = 0 << 20,
    UIViewAnimationOptionTransitionFlipFromLeft    = 1 << 20, //从左边切入
    UIViewAnimationOptionTransitionFlipFromRight   = 2 << 20, //从右边切入
    UIViewAnimationOptionTransitionCurlUp          = 3 << 20, //从上面立体进入
    UIViewAnimationOptionTransitionCurlDown        = 4 << 20, //从下面立体进入
    UIViewAnimationOptionTransitionCrossDissolve   = 5 << 20, //溶解效果
    UIViewAnimationOptionTransitionFlipFromTop     = 6 << 20, //从上面切入
    UIViewAnimationOptionTransitionFlipFromBottom  = 7 << 20, //从下面切入
};

```

UIViewAnimationOptions 枚举中实际上定义了 3 种类型的配置参数, 一种定义了视图动画执行时的一些基础属性, 例如其子视图是否执行动画、是否允许用户进行交互操作等; 一种定义了动画执行时的时间函数, 即先快后慢, 先慢后快, 匀速, 淡入淡出等; 还有一种定义了 UIView 转场动画的转场效果, 关于 UIView 的转场动画, 后面小节会进行详细介绍。

在前面小节的动画示例中, 都只执行了单独的动画效果, 使 UIView 层的动画也可以很好地支持组合动画的开发。组合动画无非两种方式, 一种是几种类型的过渡动画同时执行, 一种是几种类型的过渡动画依次执行。

关于几种类型动画同时执行的方法很简单, 在上面的方法中设置动画的 block 中同时添加多种视图属性的过渡代码即可, 如下所示。

```

-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{
    [UIView animateWithDuration:3 animations:^(
        _colorView.frame=CGRectMake(100, 100, 200, 200);
        _colorView.backgroundColor = [UIColor blueColor];
    )];
}

```

上面的代码运行效果在 3 秒内，将色块从原始位置移动到(0,0)点并且尺寸放大为(200,200)，与此同时将颜色渐变为蓝色。

关于几种类型的动画依次执行要用到 UIView 层动画嵌套的方法，示例代码如下：

```

-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{
    [UIView animateWithDuration:3 animations:^(
        _colorView.frame=CGRectMake(100, 100, 100, 100);
    )completion:^(BOOL finished) {
        [UIView animateWithDuration:2 animations:^(
            _colorView.frame = CGRectMake(100, 100, 200, 200);
        )completion:^(BOOL finished) {
            [UIView animateWithDuration:1 animations:^(
                _colorView.backgroundColor = [UIColor blueColor];
            )];
        }];
    }];
}

```

上面代码执行的效果为：先在 3 秒内将色块移动到(100,100)的位置，再在 2 秒内将色块尺寸放大为(200,200)，最后在 1 秒内将色块颜色渐变为蓝色。按原理来讲，UIView 层动画可以一直这样嵌套下去。

6.2.4 UIView 层过渡动画支持的属性

在上一小节的示例中，使用 UIView 层的相关方法进行了控件位置、尺寸、颜色的过渡动画演示。实际上除了上面进行演示的属性之外，还有一些方法支持许多 UIView 类中属性的过渡动画，如图 6-3 中所列举。

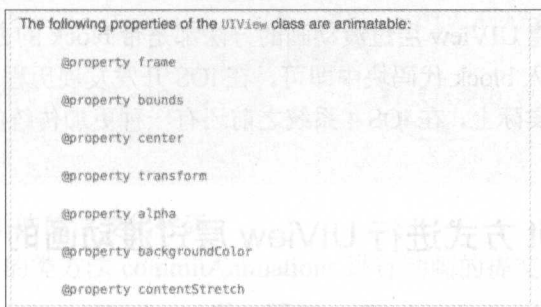


图 6-3 UIView 层动画支持的过渡属性

图 6-3 中列出的属性意义如下：

- **Frame** 为控件的位置与尺寸改变增加过渡动画效果。
- **Bounds** 为控件的内部坐标系原点改变与尺寸改变增加过渡动画。
- **Center** 为控件的中心点左边改变增加过渡动画。
- **Transform** 为控件的数学变换增加过渡动画。
- **Alpha** 为控件的透明度改变增加过渡动画。
- **backgroundColor** 为控件的背景色改变增加过渡动画。
- **contentStretch** 为 UIImageView 改变其图片的拉伸方式过程增加过渡动画。

上面介绍的属性中，**frame** 与 **bounds** 都是 **CGRect** 类型的属性，其在使用 **CGRectMake()** 方法进行设置时后两个参数的意义一样，都是设置控件的尺寸大小。但前两个参数意义却相差甚远，**frame** 属性的横纵坐标决定了控件在其父视图上的位置，是对外界而言的，**bounds** 属性的横纵坐标决定了控件内部坐标系的原点，不会影响控件对于其父视图的相对位置，但是会影响它与其内部子视图的相对位置。

控件的 **transform** 属性用于设置控件 UI 上的一些数学变化，例如翻转、旋转、平移、缩放等。例如如下代码将以动画的形式展现控件顺时针旋转 90 度。

```
[UIView animateWithDuration:3 animations:^(
    _colorView.transform = CGAffineTransformMakeRotation(M_PI_4);
    }completion:^(BOOL finished) {
    }];
```

contentStretch 属性是针对 UIImageView 而言的，在为 UIImageView 设置内容图片时，如果图片过小或者过大，UIImageView 会对图片进行整体缩放处理使其充满整个 UIImageView 控件。具体图片进行缩放的区域是由 **contentStretch** 属性设置的，这个属性也需要设置为 **CGRect** 类型，其默认为(0,0,1,1)，代表从图片的左上角到右下角都是图片拉伸缩放的区域，即对图片整体进行拉伸缩放操作。如果要设置图片中心点右侧区域和中心点下侧区域为可拉伸缩放的区域，则需将 **contentStretch** 属性设置为(0.5,0.5,1,1)。

6.3 使用 commit 方式进行 UIView 层动画的创建

在 6.2 小节中使用创建 UIView 层过渡动画的方法都是带 block 的方法，开发者只需将要执行动画的属性更改代码放入 block 代码块中即可。在 iOS 开发发展历程中，block 是在 iOS 4 系统之后才出现的新语法。实际上，在 iOS 4 系统之前还有一种更加传统的方法来创建 UIView 层的过渡动画。

6.3.1 使用 commit 方式进行 UIView 层过渡动画的创建

打开前面创建的 **UIViewAnimationTest** 工程，将 **touchesBegan:withEvent:** 中代码进行如下修改：


```

-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{
//第1部分 开始动画标志
    [UIView beginAnimations:@"test" context:nil];
//第2部分 设置动画属性
    [UIView setAnimationDelegate:self];
    [UIView setAnimationWillStartSelector:@selector(start)];
    [UIView setAnimationDidStopSelector:@selector(stop)];
    [UIView setAnimationDuration:3.0];
    [UIView setAnimationDelay:1];
    [UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];
    [UIView setAnimationRepeatAutoreverses:YES];
    [UIView setAnimationRepeatCount:3];
//第3部分 设置要执行动画的属性
    _colorView.backgroundColor = [UIColor blueColor];
//第4部分 提交动画
    [UIView commitAnimations];
}

```

使用 commit 方式进行 UIView 层过渡动画的创建主要分为 4 个部分，绝大部分代码都由 UIView 的类方法提供。

首先，使用 UIView 的类方法 beginAnimations:context: 标记动画开始，这个方法中第 1 个参数设置此动画动作的标识符。在此方法与 commitAnimations 方法之间的代码为动画的参数配置与具体要执行的动画动作。

第 2 部分为动画的配置部分，这部分通过一系列 UIView 的类方法来对 UIView 层的属性过渡动画进行参数配置。setAnimationDelegate: 方法设置接收动画开始于结束消息的代理类对象，setAnimationWillStartSelector: 方法设置动画开始时回调的代理方法，setAnimationDidStopSelector: 方法设置动画结束后回调的代理方法。setAnimationDuration: 方法设置动画执行的时间。setAnimationDelay: 方法设置动画延时多少秒后开始执行。setAnimationRepeatAutotrverses: 方法设置动画是否自动逆向执行，这个方法中的参数设置为 YES，则当动画顺向执行完毕后会 自动逆向执行一次。setAnimationRepeatCount: 方法设置动画执行的循环次数。setAnimationCurve: 方法设置动画执行的时间函数类型，其枚举及意义如下所示。

```

typedef NS_ENUM(NSInteger, UIViewAnimationCurve) {
    UIViewAnimationCurveEaseInOut,    //淡入淡出    UIViewAnimationCurveEaseIn,
                                        // 仅仅淡入
    UIViewAnimationCurveEaseOut,      //仅仅淡出
    UIViewAnimationCurveLinear        //线性
};

```

第 3 部分是执行动画的属性变化代码。

最后，调用 UIView 的类方法 commitAnimations 进行动画的提交，调用此方法之后，过渡动画开始正式执行。



提示

1. 上述过渡动画的 4 个部分顺序不能颠倒。
2. 如果不调用 `setAnimationDelegate:` 方法进行代理的设置，使用 `setAnimationWillStartSelector:` 与 `setAnimationDidStopSelector:` 方法设置的回调方法将无效。

6.3.2 两种 UIView 层动画创建方式的优劣

使用 `commit` 方式创建过渡动画的方法更加传统，兼容性也更加好一些，但是毋庸置疑，目前几乎所有 iOS 设备的版本都在 iOS 4 版本之上，实际上在应用中基本不存在这方面的兼容问题。从上面的代码进行比较也可以发现，使用 `block` 的方式和使用 `commit` 的方式完成相同的动画效果，`commit` 的方式更加复杂，代码量也偏多。如果进行组合动画的开发，`block` 方式的优势就会更加明显了。因此，无论从方便开发者角度还是官方推荐的角度，使用 `block` 的方式进行 UIView 层过渡动画的开发都是开发者的首选。

6.4 UIView 的转场动画

相对于过渡动画转场动画更多用于两个 UIView 视图的切换或者重绘某个 UIView 视图。在 iOS 开发中，UIView 的转场动画有两种，一种是对 UIView 视图内容进行重绘而不切换视图时使用的转场效果，一种是用新的 UIView 视图代替旧的 UIView 视图时使用的转场动画。

6.4.1 重绘 UIView 视图时使用的转场动画

在实际开发中，开发者时常会遇到这样的需求：一个 UIView 视图中有许多子视图控件，当用户交互产生数据变化时，这些子视图控件需要进行重新布局等操作，使原 UIView 视图看起来好像换成了另一个。实际上视图的重绘是在瞬间完成的，如果不加任何转场效果，闪屏会使用户感觉非常突兀，这时 UIView 的转场动画就派上了用场。

使用 Xcode 创建一个名为 `UIViewTransitionTest` 的工程，在 `ViewController.m` 文件中声明一个 UIView 属性作为演示色块。

```
@interface ViewController ()
{
    UIView * _contentView;
}
@end
```

在 `ViewController.m` 文件的 `viewDidLoad` 中编写的初始化代码如下。

```
- (void)viewDidLoad {
    [super viewDidLoad];
```

```

    _contentView = [[UIView alloc] initWithFrame:CGRectMake(100, 100, 100,
100)];
    _contentView.backgroundColor = [UIColor redColor];
    [self.view addSubview:_contentView];
}

```

在 ViewController.m 文件中实现 touchesBegan:withEvent: 方法进行动画效果的演示。

```

- (void)touchesBegan: (NSSet<UITouch *> *)touches withEvent: (UIEvent *)event{
    [UIView transitionWithView:_contentView duration:3 options:UIViewAnim
ationOptionTransitionFlipFromBottom|UIViewAnimationOptionAllowAnimatedConten
t animations:^(
        _contentView.backgroundColor = [UIColor blueColor];
    } completion:nil];
}

```

UIView 的类方法 transitionWithView:duration:options:animation:completion: 用于完成视图重绘方式的转场动画, 这个方法中第 1 个参数为要执行转场动画的 UIView 对象。第 2 个参数设置动画执行的时间。第 3 个参数设置动画执行的一些配置参数, 其中可用的枚举在 5.1 小节中有介绍。第 4 个参数为要执行的重绘动作, 开发者可以在这个 block 中进行改变视图控件的属性的操作或者子视图的重新布局。实际上, 就算这个 block 中不编写任何重绘操作代码, 视图的转场动画依然会执行, 只是执行后的视图没有任何变化。第 5 个参数设置动画执行完成后的回调 block。UIView 的转场动画也支持组合嵌套。

6.4.2 切换 UIView 视图时使用的转场动画

切换类型的转场动画用于开发中整体视图的切换需求, 这一操作会将原先的 UIView 视图从其父视图上移除, 然后将新的 UIView 视图代替原视图添加在父视图上。

打开 UIViewTransitionTest 工程, 在 ViewController.m 文件中再声明一个 UIView 属性作为要切换的视图控件。

```

@interface ViewController ()
{
    UIView * _contentView;
    UIView * _contentView2;
}

```

在 viewDidLoad 方法中添加 _contentView2 的初始化代码。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    _contentView = [[UIView alloc] initWithFrame:CGRectMake(100, 100, 100,
100)];
    _contentView.backgroundColor = [UIColor redColor];
    [self.view addSubview:_contentView];
}

```



```

_contentView2 = [[UIView alloc] initWithFrame:CGRectMake(100, 100, 100,
100)];
_contentView2.backgroundColor = [UIColor blueColor];
}

```

上面只对 _contentView2 进行初始化，并不将其添加到界面视图上。

在 touchesBegan:withEvent: 方法中添加如下代码：

```

-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{
    [UIView transitionFromView:_contentView toView:_contentView2 duration:
3 options:UIViewAnimationOptionTransitionFlipFromBottom|UIViewAnimationOptio
nAllowAnimatedContent completion:nil];}

```

transitionFromView:toView:options:completion: 方法用于创建切换视图的转场动画，这个方法中第 1 个参数设置当前显示的 View 视图，第 2 个参数设置将要切换的 View 视图，第 3 个参数设置执行动画的配置参数，第 4 个参数为动画执行完毕后的回调 block 代码块。

在执行切换视图转场动画时，实际上将原视图从其父视图上移除掉，再添加上新的视图，切换视图的动画也是作用在两个切换视图的父视图上的。

6.5 核心动画编程技术——CoreAnimation

在前面章节中介绍了有关 UIView 层的过渡动画与转场动画的应用，使用 UIView 类有关动画的类方法基本可以满足开发中遇到的大部分需求。然而，UIView 层的动画仍然有许多局限性，开发者若想更加自由地进行 iOS 动画编程还需要使用一个更加底层也更加强大的框架：CoreAnimation。

CoreAnimation 框架也被称为核心动画编程框架，它是基于 OpenGL 与 CoreGraphics 图像处理框架的一个跨平台的动画框架。如图 6-4 所示描述了 CoreAnimation 框架的系统结构。

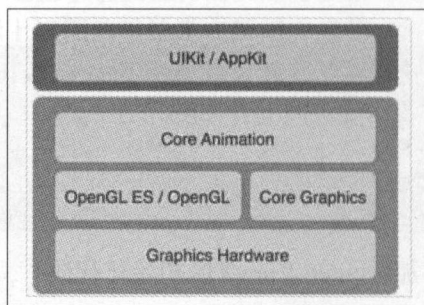


图 6-4 CoreAnimation 框架的系统结构

CoreAnimation 框架中大多数的动画效果都是基于 CALayer 类来实现的。任何一个 UIView 视图都包含了一个 CALayer 属性，CALayer 对象负责 UIView 视图的渲染与层级结构。其实 CALayer 对读者来说并不陌生，在第 2 章中进行视图控件的圆角、边框、阴影等操作的时候就已经使用了 Layer 层的属性。

6.5.1 锚点对视图控件几何位置的影响

CoreAnimation 的动画开发是基于 Layer 层的, 要了解 CALayer 类的使用, 首先应该先了解一个概念: 锚点。

所有的 CALayer 对象都有一个 anchorPoint 属性, 这个属性就是 CALayer 对象的锚点, 是 CGPoint 类型。锚点可以理解成对象动作的参照点, 其 x, y 值的取值范围都是 0~1。CALayer 对象默认的锚点值为(0,0), 即 Layer 层的左上角, 如果要将锚点设置 Layer 层的中心, 则可以设置 anchorPoint 的值为(0.5,0.5)。当 CALayer 层进行动作操作时, 执行的动作将以锚点作为参照点, 例如进行旋转、平移等操作。图 6-5 与图 6-6 所示为设置不同位置锚点的 CALayer 对象。

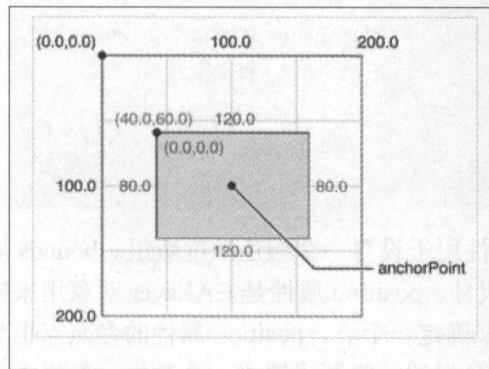


图 6-5 设置锚点为(0.5, 0.5)

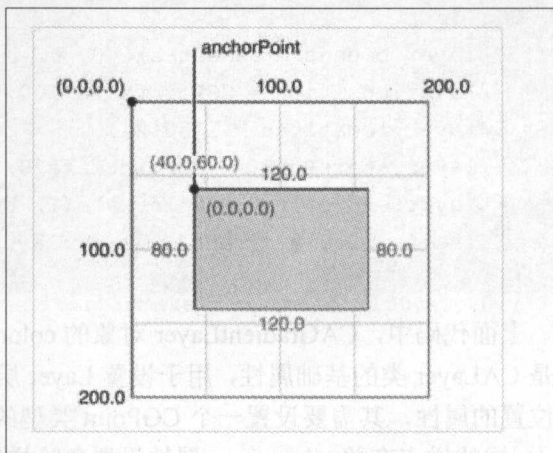


图 6-6 设置锚点为(0,0)

当对图 6-5 与图 6-6 中的视图进行旋转操作时, 效果如图 6-7 与图 6-8 所示。

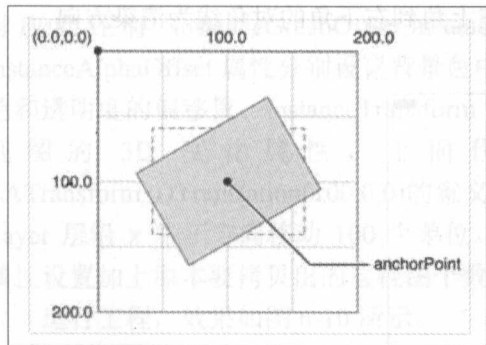


图 6-7 以视图中心为锚点进行旋转

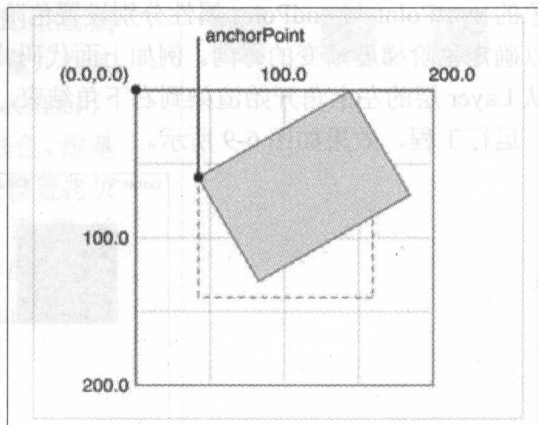


图 6-8 以视图左上角为锚点进行旋转

当一个 UIView 对象被创建出来后, 其内部自带一个 CALayer 对象, Layer 层中主要保存视图的绘制信息, View 层则除了整合视图的绘制外, 还封装了接收事件与处理用户交互的功能。

6.5.2 色彩梯度层——CAGradientLayer

CAGradientLayer 类是继承于 CALayer 类的子类,专门用来创建颜色梯度渐变的视图效果。

使用 Xcode 创建一个名为 CAGradientLayerTest 的工程,在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    CAGradientLayer * layer = [CAGradientLayer layer];
    layer.colors = @[(id)[UIColor redColor].CGColor, (id)[UIColor blueColor].CGColor];
    layer.bounds = CGRectMake(0, 0, 100, 100);
    layer.position = CGPointMake(100, 100);
    layer.locations = @[@0.2];
    layer.startPoint = CGPointMake(0, 0);
    layer.endPoint = CGPointMake(1, 1);
    [self.view.layer addSublayer:layer];
}
```

上面代码中, CAGradientLayer 对象的 colors 属性用于设置一个颜色梯度数组。bounds 属性是 CALayer 类的基础属性,用于设置 Layer 层的尺寸。position 属性是 CALayer 对象用来确定位置的属性,其需要设置一个 CGPoint 类型的值来确定一个点,position 属性的参照点也与 Layer 层的锚点有关。locations 属性设置色阶梯度的分界线,需要设置成一个数组,数组中元素均为 0~1 之间并且递增的 NSNumber 类型的对象,例如上面代码将其色阶梯度分界线设置为 0.2,则红色会渲染 Layer 层 20%尺寸后才开始进行向蓝色渐变的梯度变化。CAGradientLayer 对象的 startPoint 与 endPoint 属性分别设置色阶梯度渲染的起始点与结束点,通过这两个属性可以确定色阶梯度渐变的方向。例如上面代码将起始点设置为(0,0),结束点设置为(1,1),则颜色从 Layer 层的左上角开始渲染到右下角结束,按左上角到右下角的对角线为渲染方向。

运行工程,效果如图 6-9 所示。

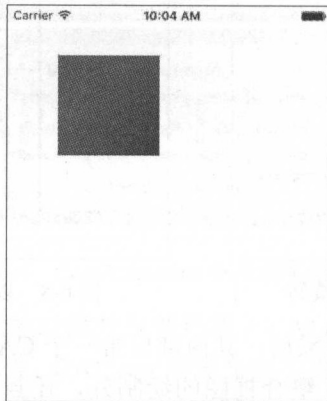


图 6-9 使用 CAGradientLayer 类创建的色阶梯度视图层

6.5.3 视图拷贝层——CAReplicatorLayer

CAReplicatorLayer 类可以理解为一个 Layer 层拷贝容器，其作用并非是进行具体某一个的 UI 展现，而是拷贝一个已经存在的 Layer 层对象进行复制渲染。

使用 Xcode 创建一个名为 CAReplicatorLayerTest 的工程，在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    //创建一个普通 layer
    CALayer * layer = [CALayer layer];
    layer.bounds = CGRectMake(0, 0, 50, 50);
    layer.position = CGPointMake(50, 100);
    layer.backgroundColor = [UIColor redColor].CGColor;
    //创建拷贝容器
    CAReplicatorLayer * reLayer = [CAReplicatorLayer layer];
    reLayer.instanceRedOffset = -0.2;
    reLayer.position = CGPointMake(0, 0);
    reLayer.instanceTransform = CATransform3DMakeTranslation(100, 0, 0);
    reLayer.instanceCount = 3;
    [reLayer addSublayer:layer];
    [self.view.layer addSublayer:reLayer];
}
```

上面代码中先创建了一个普通的 Layer 层作为要拷贝的原本，将其作为子 Layer 添加进 CAReplicatorLayer 对象中。CAReplicatorLayer 对象的 instanceRedOffset 属性设置每个拷贝的副本背景色中红基色的偏移量，与其对应的还有 instanceGreenOffset、instanceBlueOffset，instanceAlphaOffset 属性分别设置背景色中蓝基色、绿基色和透明度的偏移量。instanceTransform 属性设置拷贝视图的 3D 变化属性，上面代码设置的 CATransform3DTranslation(100,0,0) 的意义为将每个拷贝 Layer 层沿 x 轴正方向移动 100 个单位。instanceCount 属性设置加上原本要拷贝出的层视图个数。

运行工程，效果如图 6-10 所示。

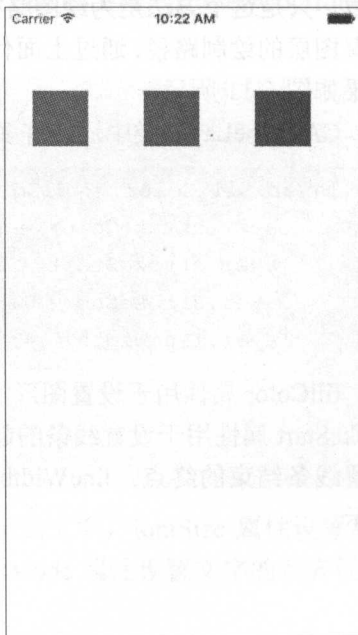


图 6-10 使用 CAReplicatorLayer 类创建的 Layer 拷贝视图层

6.5.4 图形渲染层——CAShapeLayer

CAShapeLayer 是用于绘制图形的 Layer 层的，开发者可以使用它进行自定义的图形绘制。

使用 Xcode 创建一个名为 CAShapeLayerTest 的工程，在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    CAShapeLayer * layer = [CAShapeLayer layer];
    layer.position = CGPointMake(0, 0);
    CGMutablePathRef path = CGPathCreateMutable();
    CGPathMoveToPoint(path, 0, 100, 100);
    CGPathAddLineToPoint(path, 0, 300, 100);
    CGPathAddLineToPoint(path, 0, 200, 200);
    CGPathAddLineToPoint(path, 0, 100, 100);
    layer.path=path;
    [self.view.layer addSublayer:layer];
}
```

上面代码中，CGMutablePathRef 是 iOS 中的绘图路径对象，CGPathCreateMutable()方法用于创建可变的绘图路径对象，CGPathMoveToPoint()方法设置绘图路径的起点，这个方法中第 1 个参数为绘图路径对象，第 2 个参数为 transform 变化参数，第 3 个参数为 x 点坐标，第 4 个参数为 y 点坐标。CGPathAddLineToPoint()方法中参数的意义与 CGPathMoveToPoint()方法一致，只是这个方法是为绘图路径对象添加一条绘图线。CAShapeLayer 对象的 path 属性用于设置图层的绘制路径，通过上面代码的设置将在屏幕上绘制一个三角形的 Layer 层，运行工程，效果如图 6-11 所示。

CAShapeLayer 类中还有许多方法用来绘制的图案进行自定义设置，示例代码如下：

```
layer.fillColor = [UIColor redColor].CGColor;
layer.strokeColor = [UIColor blueColor].CGColor;
layer.strokeStart = 0.3;
layer.strokeEnd = 0.8;
layer.lineWidth = 4;
```

fillColor 属性用于设置图形的填充颜色，strokeColor 属性用于设置图形的边框线条颜色，strokeStart 属性用于设置线条的起点，这个值是一个比例值，为占周长的比例，strokeEnd 属性设置线条结束的终点，lineWidth 属性设置边框线条的宽度。再次运行工程，效果如图 6-12 所示。

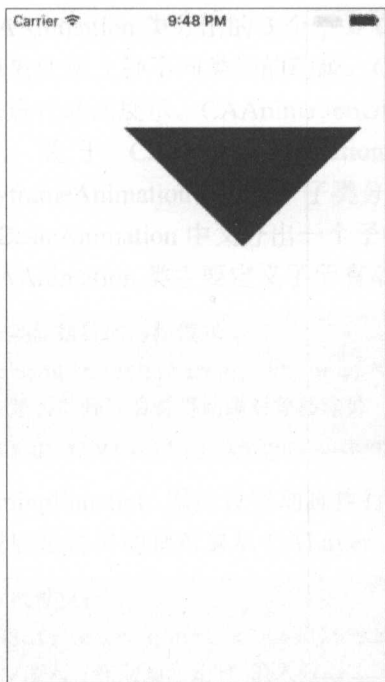


图 6-11 CAShapeLayer 层绘制的三角形

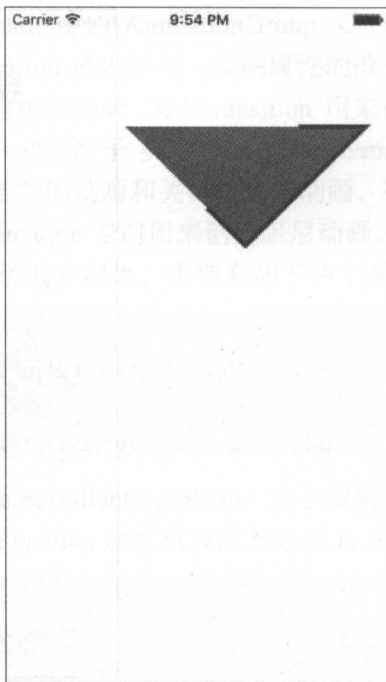


图 6-12 为 CAShapeLayer 绘制的图形进行自定义设置

6.5.5 文本绘制层——CATextLayer

CATextLayer 用于进行视图上文本的绘制，UIKit 框架中最基础的 UILabel 控件就是基于 CATextLayer 实现的。使用 Xcode 创建一个名为 CATextLayerTest 的工程，在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    CATextLayer * layer = [CATextLayer layer];
    layer.bounds = CGRectMake(0, 0, 320, 100);
    layer.position = CGPointMake(160, 100);
    layer.string = @"我是文字 Layer";
    layer.fontSize = 25;
    layer.foregroundColor = [UIColor redColor].CGColor;
    layer.alignmentMode = kCAAlignmentCenter;
    [self.view.layer addSublayer:layer];
}
```

上面代码中，CATextLayer 的 string 属性用于设置要显示的文字，fontSize 属性设置显示文字的字号，foregroundColor 属性设置文字的颜色，alignmentMode 属性设置文字的对齐模式。运行工程，效果如图 6-13 所示。

除了上面介绍的这些 Layer 层类，CALayer 家族中还有一个十分强大的类 CAEmitterLayer，专门用来处理一些粒子效果生成的动画，因其比较复杂，将安排在后面章节单独进行介绍。

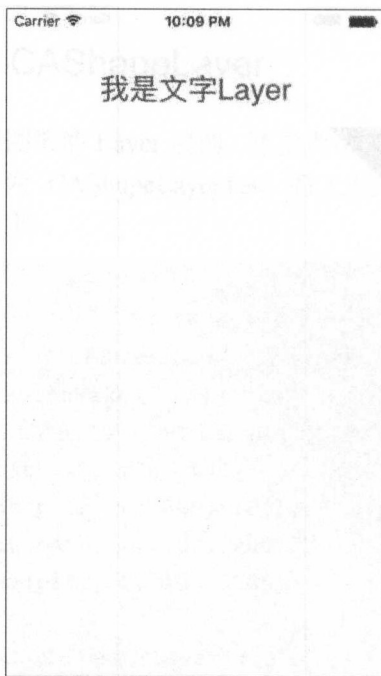


图 6-13 CATextLayer 创建的渲染文本的 Layer 层

6.5.6 CAAAnimation 动画体系介绍

通过前面几个小节介绍，读者可以了解到，通过 CALayer 开发者可以设置许多视图 UI 上的属性，CAAnimation 框架的作用就是将这些属性的变化都以动画的形式展现出来。由于 CALayer 相对于 UIView 可以更灵活地设置更多属性，所以对于 CALayer 层的动画操作也会更加灵活自由。

CoreAnimation 框架中的核心类是 CAAnimation，CAAnimation 是动画操作的基类，开发者主要使用一些继承自它的子类来进行 CALayer 层动画的开发。如图 6-14 所示为 CAAnimation 及其子类的关系结构图。

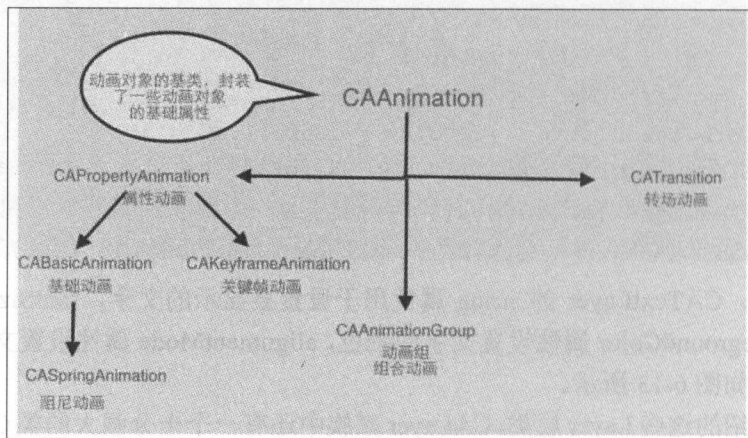


图 6-14 CAAnimation 及其子类结构图

CAAnimation 类分出的 3 个子类 CAPROPERTYAnimation、CAAnimationGroup、CATransition 分别用来处理 3 种不同类型的动画。CAPROPERTYAnimation 最为常用，视图属性的改变都可以由它来进行动画展示。CAAnimationGroup 类用于创建组合动画。CATransition 用来创建转场动画。关于 CAPROPERTYAnimation，它又分出两个子类，CABasicAnimation 和 CAKeyframeAnimation，这两个子类分别用来处理基础类的动画和关键帧类的动画。在基础动画 CABasicAnimation 中又分出一个子类 CASpringAnimation 专门用来创建阻尼动画。

CAAnimation 类主要定义了所有动画操作都会有的共有属性，主要有如下两个属性：

```
//动画执行的时序模式
@property(nullable, strong) CAMediaTimingFunction *timingFunction;
//是否动画完成时将动画对象移除掉
@property(getter=isRemovedOnCompletion) BOOL removedOnCompletion;
```

timingFunction 属性设置动画执行的时间函数，removedOnCompletion 用于设置当动画播放完成后是否将动画对象从 CALayer 上除掉。timingFunction 属性可设的参数值如下所示：

```
//线性执行
NSString * const kCAMediaTimingFunctionLinear;
//淡入 在动画开始时 淡入效果
NSString * const kCAMediaTimingFunctionEaseIn;
//淡出 在动画结束时 淡出效果
NSString * const kCAMediaTimingFunctionEaseOut;
//淡入淡出
NSString * const kCAMediaTimingFunctionEaseInEaseOut;
//默认效果
NSString * const kCAMediaTimingFunctionDefault;
```

使用如下的示例代码方法进行 CAMediaTimingFunction 对象的创建。

```
[CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionDefault];
```

CAPROPERTYAnimation 类中封装了用于属性变化的动画的基类属性，主要代码如下：

```
//创建初始化对象方法
+ (instancetype)animationWithKeyPath:(nullable NSString *)path;
//这个属性确定动画执行的状态是否叠加在控件的原状态上
@property(getter=isAdditive) BOOL additive;
//这个属性对重复执行的动画有效果
@property(getter=isCumulative) BOOL cumulative;
//这个属性和 transfrom 属性的动画执行相关
@property(nullable, strong) CAValueFunction *valueFunction;
```

在上面的方法和属性中，animationWithKeyPath:方法用于创建并初始化动画对象，其中 path 参数即为要执行动画的属性。例如如果要执行背景色渐变的动画，则 path 可设置为 @"backgroundColor"。additive 设置动画执行的状态是否叠加，如果设置为 NO，当执行移动动画时，如果执行两次，两次动画都会从视图的原始位置进行移动，如果设置 YES，则第 2 次

移动动画的起点会从第 1 次移动的终点开始。cumulative 属性与重复执行的动画有关，如果设置为 NO，则重复动画每次执行都会从原始状态开始执行，设置为 YES 则重复动画的每次效果都会叠加。ValueFunction 这个属性比较特殊，其专门用来处理 transform 变换类的动画，在后面小节的应用中会向读者详细介绍这个属性的应用。

6.5.7 使用 CABasicAnimation 创建基础动画

前面小节介绍了许多基础知识，从 CALayer 层的基本用法到 CAAAnimation 的基类属性。从本节开始将进行一些 CoreAnimation 动画开发的实际操练。CABasicAnimation 是 CoreAnimation 动画框架中最常用的动画执行类。使用 Xcode 创建一个名为 CABasicAnimationTest 的工程，在 ViewController.m 文件中声明一个 CALayer 类的对象，如下所示：

```
@interface ViewController ()
{
    CALayer * layer;
}
@end
```

在 viewDidLoad 方法中进行 layer 属性的创建和初始化相关操作，如下所示：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    layer = [CALayer layer];
    layer.bounds = CGRectMake(0, 0, 100, 100);
    layer.position = CGPointMake(160, 100);
    layer.backgroundColor = [UIColor redColor].CGColor;
    [self.view.layer addSublayer:layer];
}
```

在 ViewController.m 文件中实现 touchesBegan:withEvent: 方法，如下所示：

```
-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{
    //背景色渐变动画
    CABasicAnimation * ani = [CABasicAnimation animationWithKeyPath:@"back
groundColor"];
    //从红色开始
    ani.fromValue = (id)[UIColor redColor].CGColor;
    //渐变成蓝色
    ani.toValue = (id)[UIColor blueColor].CGColor;
    //时间 2S
    ani.duration = 2;
    //执行动画
    [layer addAnimation:ani forKey:@""];
}
```


上面代码设置为背景色的变化添加渐变动画, `fromValue` 属性设置起始的颜色。 `toValue` 属性设置要渐变成的颜色。 `duration` 属性设置动画执行的时间。调用 `CALayer` 类的 `addAnimation:forKey:` 方法进行动画动作的添加和执行, 这个方法中第 1 个参数为要执行的动画动作对象, 第 2 个参数将为这个动作设置一个标识符。



提示

`CABasicAnimation` 类中还有 `byValue` 这样一个属性, `fromValue` 和 `byValue` 两个属性值的差异如下所示。

- `fromValue` 和 `toValue` 不为空: 动画的值由 `fromValue` 变化到 `toValue`
- `fromValue` 和 `byValue` 不为空: 动画的值由 `fromValue` 变化到 `fromValue+byValue`
- `byValue` 和 `toValue` 不为空: 动画的值由 `toValue-byValue` 变化到 `toValue`
- 只有 `fromValue` 不为空: 动画的值由 `fromValue` 变化到 layer 的当前状态值
- 只有 `toValue` 不为空: 动画的值由 layer 当前的值变化到 `toValue`
- 只有 `byValue` 不为空: 动画的值由 layer 当前的值变化到 layer 当前的值 + `byValue`

如果要执行 `transform` 变换的相关动画, 例如使视图沿 `z` 轴进行旋转(即在屏幕平面旋转), 使用如下的代码。

```
-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{
    //绕 z 轴旋转的动画
    CABasicAnimation * ani = [CABasicAnimation animationWithKeyPath:@"transform"];
    //从 0 度开始
    ani.fromValue = @0;
    //旋转到 180 度
    ani.toValue = [NSNumber numberWithFloat:M_PI];
    //时间 2s
    ani.duration = 2;
    //设置为 z 轴旋转
    ani.valueFunction = [CAValueFunction functionName:kCAValueFunctionRotateZ];
    //执行动画
    [layer addAnimation:ani forKey:@""];
}
```

上面在创建 `CABasicAnimation` 对象时, 设置其 `KeyPath` 为 `transform`。但是具体的变换方式是由 `valueFunction` 属性决定的, `[CAValueFunction functionName:kCAValueFunctionRotateZ]` 创建沿 `z` 轴旋转的动画方式, 还可以通过下面的创建参数进行其他 `transform` 变换方式的设置:

```

//设置沿 x 轴旋转操作
kCAValueFunctionRotateX
//设置沿 y 轴旋转操作
kCAValueFunctionRotateY
//设置沿 z 轴旋转操作
kCAValueFunctionRotateZ
//设置沿 x 轴缩放操作
kCAValueFunctionScaleX
//设置沿 y 轴缩放操作
kCAValueFunctionScaleY
//设置沿 z 轴缩放操作
kCAValueFunctionScaleZ
//设置沿 x 轴平移操作
kCAValueFunctionTranslateX
//设置沿 y 轴平移操作
kCAValueFunctionTranslateY
//设置沿 z 轴平移操作
kCAValueFunctionTranslateZ

```

CASpringAnimation 动画和 CABasicAnimation 动画相比只是多了一些关于阻尼操作的参数，开发者可以设置这些参数在动画的展现过程中添加类似弹簧的阻尼效果，可用属性如下所示。

```

//这个属性设置类似弹簧重物的质量，会影响惯性，所以必须大于 0 默认为 1
@property CGFloat mass;
//设置弹簧的刚度系数，必须大于 0，默认为 100，这个越大则回弹越快
@property CGFloat stiffness;
//阻尼系数默认为 10，必须大于 0，这个值越大回弹的幅度越小
@property CGFloat damping;
//初始速度
@property CGFloat initialVelocity;
//获取动画停下来需要的时间 只读属性
@property(readonly) CTimeInterval settlingDuration;

```

6.5.8 使用 CAKeyframeAnimation 类创建关键帧动画

CAKeyframeAnimation 动画也被称为关键帧动画，也是继承自 CAPropertyAnimation，用于创建 Layer 层属性变化相关的动画。比 CABasicAnimation 更加强大的是，CABasicAnimation 创建的属性变化动画只能通过 fromValue、byValue 和 toValue 这些属性设置 Layer 属性变化的起始值和结束值。而 CAKeyframeAnimation 可以通过设置动画关键帧的方式自由控制整个动画的过程。

使用 Xcode 创建一个名为 CAKeyframeAnimationTest 的工程，在 ViewController.m 文件中声明一个 CALayer 类型的属性。

```
@interface ViewController ()
{
    CALayer * _layer;
}
@end
```

在 `viewDidLoad` 方法中进行相关的初始化操作，如下所示。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    _layer = [CALayer layer];
    _layer.bounds = CGRectMake(0, 0, 100, 100);
    _layer.backgroundColor = [UIColor redColor].CGColor;
    _layer.position = CGPointMake(50, 100);
    [self.view.layer addSublayer:_layer];
}
```

在 `ViewController.m` 文件中实现 `touchesBegan:withEvent:` 方法进行效果的测试。

```
-(void)touchesBegan:(NSSet<UITouch * > *)touches withEvent:(UIEvent *)event{
    CAKeyframeAnimation * ani = [CAKeyframeAnimation animationWithKeyPath:
@"position"];
    ani.values = @[ [NSValue valueWithCGPoint:CGPointMake(50, 100)], [NSValue
valueWithCGPoint:CGPointMake(120, 100)], [NSValue valueWithCGPoint:CGPointMa
ke(120, 200)], [NSValue valueWithCGPoint:CGPointMake(200, 200)]];
    ani.keyTimes = @[@0, @0.5, @0.8, @1];
    ani.duration = 3;
    [_layer addAnimation:ani forKey:@""];
}
```

在上面的代码中，`CAKeyframeAnimation` 类的 `values` 属性用于设置关键帧数组，例如上面进行的动画动作为 `position` 位置的移动，在这个数组中需要设置为动画过程中的位置转折点。`CAKeyframeAnimation` 类的 `keyTimes` 属性设置每段动画的时间占比，其中值的取值范围为 0~1 之间并且是递增的。

6.5.9 CALayer 层的转场动画——CATransition

`CATransition` 动画用来处理 `CALayer` 层的转场效果，和 `CAPropertyAnimation` 动画不同之处在于 `CAPropertyAnimation` 动画是当 `CALayer` 对象的属性改变时展示动画效果，而 `CATransition` 是当 `CALayer` 对象出现时展现动画效果。

使用 Xcode 创建一个名为 `CATransitionTest` 的工程，直接在 `ViewController.m` 文件中实现如下方法。

```
-(void)touchesBegan:(NSSet<UITouch * > *)touches withEvent:(UIEvent *)event{
    CALayer * layer = [CALayer layer];
```



```

layer.bounds = CGRectMake(0, 0, 100, 100);
layer.position = CGPointMake(100, 100);
layer.backgroundColor = [UIColor redColor].CGColor;
CATransition * ani = [CATransition animation];
ani.type = kCATransitionPush;
ani.subtype = kCATransitionFromRight;
ani.duration = 3;
[layer addAnimation:ani forKey:@""];
[self.view.layer addSublayer:layer];
}

```

在上面的代码中，CATransition 类的 type 属性设置转场动画的类型，可选的参数字符串如下所示。

```

//淡入
NSString * const kCATransitionFade;
//移入
NSString * const kCATransitionMoveIn;
//压入
NSString * const kCATransitionPush;
//溶解
NSString * const kCATransitionReveal;

```

CATransition 类的 subtype 属性设置动画执行的方向，可选的参数字符串如下所示。

```

//从右侧进
NSString * const kCATransitionFromRight;
//从左侧进
NSString * const kCATransitionFromLeft;
//从上侧进
NSString * const kCATransitionFromTop;
//从下侧进
NSString * const kCATransitionFromBottom;

```



提示

关于 CATransition 类中 type 属性定义的动画效果，除了官方文档中定义的上图 4 种外，还有一些私有的类型可以使用，需要注意，使用私有的动画类型在提交 AppStore 时很有可能会被拒绝。可用的私有动画类型如下所示：

- pageCurl 翻页效果动画
- rippleEffect 滴水效果动画
- suckEffect 收缩效果，如一块布被抽走
- cube 立方体效果
- oglFlip 上下翻转效果

6.5.10 CALayer 层的组合动画——CAAnimationGroup

CAAnimationGroup 类并没有定义特定的动画类型，可以把它理解为一个动画容器，通过 CAAnimationGroup 可以将上面小节所介绍的动画效果进行组合展示。

使用 Xcode 创建一个名为 CAAnimationGroupTest 的工程，在 ViewController.m 中声明如下属性。

```
@interface ViewController ()
{
    CALayer * _layer;
}
@end
```

在 viewDidLoad 中作如下初始化操作。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    _layer = [CALayer layer];
    _layer.bounds= CGRectMake(0, 0, 100, 100);
    _layer.position = CGPointMake(100, 100);
    _layer.backgroundColor = [UIColor redColor].CGColor;
    [self.view.layer addSublayer:_layer];
}
```

在 ViewController.m 文件中实现 touchesBegan:withEvent:方法。

```
-(void)touchesBegan: (NSSet<UITouch *> *)touches withEvent: (UIEvent *)event{
    CABasicAnimation * ani1 = [CABasicAnimation animationWithKeyPath:@"backgroundColor"];
    ani1.toValue = (id) [UIColor blueColor].CGColor;
    CABasicAnimation * ani2 = [CABasicAnimation animationWithKeyPath:@"position"];
    ani2.toValue = [NSValue valueWithCGPoint:CGPointMake(200, 300)];
    CAAnimationGroup * group = [CAAnimationGroup animation];
    group.duration = 3;
    group.animations = @[ani1,ani2];
    [_layer addAnimation:group forKey:@""];
}
```

CAAnimationGeoup 中的 animations 数组中需要设置为 CAAnimation 的对象，设置的动画效果将会被同时组合展示，例如上面代码会将色块在 3 秒内位置移动到(200,300)的点并且颜色渐变为蓝色。

6.5.11 CATransform3D 变换的应用

任何 CALayer 对象都有 transform 这样一个属性，这个属性是用来设置 CALayer 对象视图渲染的数学变换效果的。在前面章节中有示例，transform 属性是可以支持 CoreAnimation 动画效果的。CATransform3D 是一个四维矩阵的结构体，可以实现视图的平移、旋转、景深旋转、缩放、镜像翻转等操作。

使用 Xcode 创建一个名为 CATransform3DTest 的工程，为了便于演示效果，向工程中添加一张图片素材。

在 ViewController.m 中的 viewDidLoad 方法中添加如下代码实现视图的平移变换。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIImageView * image1 = [[UIImageView alloc] initWithFrame:CGRectMake(10
0, 100, 100, 100)];
    image1.image = [UIImage imageNamed:@"image"];
    UIImageView * image2 = [[UIImageView alloc] initWithFrame:CGRectMake(10
0, 100, 100, 100)];
    image2.image = [UIImage imageNamed:@"image"];
    CATransform3D trans = CATransform3DTranslate(image2.layer.transform, 1
00, 100, 0);
    image2.layer.transform = trans;
    [self.view addSubview:image1];
    [self.view addSubview:image2];
}
```

在上面的代码中创建了两个一样的 UIImageView 对象，并且将其设置在相同的位置，对 image2 对象进行了平移变换，CATransform3DTranslate()方法用于创建平移变化，这个方法第 1 个参数为原 CATransform3D 对象，第 2 个参数为平移的 x 坐标值，第 3 个参数为平移的 y 坐标值，第 4 个参数为平移的 z 坐标值。运行工程，效果如图 6-15 所示。

使用如下代码进行缩放变换。

```
CATransform3D tran = CATransform3DScale(trans, 0.5, 2, 0);
```

上面方法第 1 个参数为原始的 CATransform3D 对象，第 2 个参数为 x 轴方向的缩放比，第 3 个参数为 y 轴方向的缩放比，第 4 个参数为 z 轴方向的缩放比。运行工程，效果如图 6-16 所示。

使用下面代码进行旋转变换。

```
CATransform3D tran = CATransform3DRotate(trans, M_PI_4, 0, 0, 1);
```

上面方法中第 1 个参数为原始的 CATransform3D 对象，第 2 个参数为要旋转的角度，为弧度制。第 3 个参数为旋转方向在 x 轴上的分量，第 4 个参数为旋转方向在 y 轴上的分量，第 5 个参数为旋转方向，在 z 轴上的分量，运行工程，效果如图 6-17 所示。

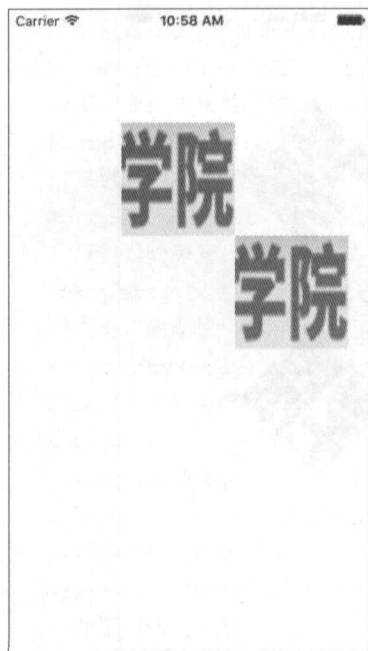


图 6-15 transform 平移变换

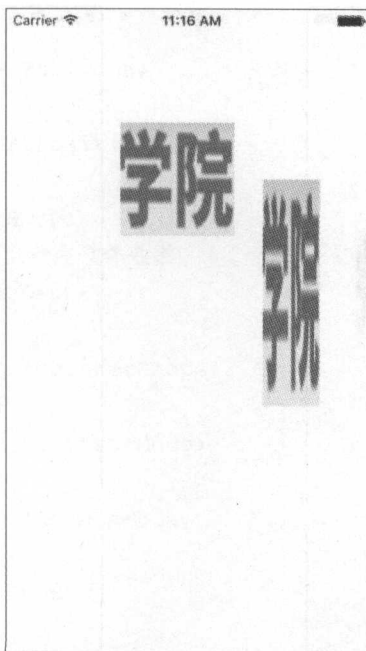


图 6-16 进行 transform 缩放变换

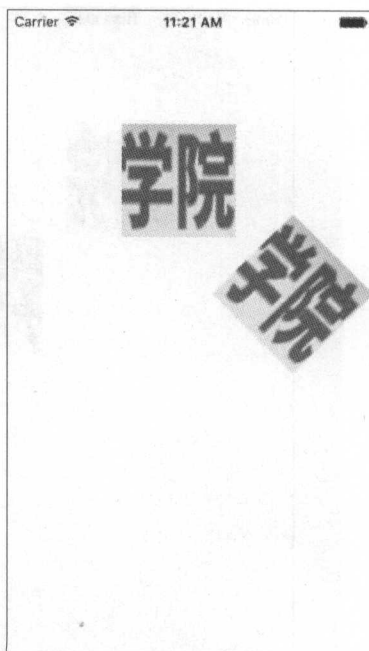


图 6-17 进行 transform 旋转变换

使用下面方法进行带景深效果的旋转变换。

```
//平移
CATransform3D trans = CATransform3DTranslate(image2.layer.transform, 100, 100, 0);
trans.m34 = -1/600.0;
image2.layer.transform = CATransform3DRotate(trans, M_PI_4, 0, 1, 0);
```

CATransform3D 结构的 m34 参数设置景深的参数，运行工程，效果如图 6-18 所示。

下面方法进行旋转镜像变换。

```
CATransform3D tran = CATransform3DRotate(image2.layer.transform, M_PI_4, 0, 0, 1);
image1.layer.transform = tran;
image2.layer.transform = CATransform3DInvert(tran);
```

CATransform3DInvert 方法用于将一个变换效果进行翻转，运行效果如图 6-19 所示。



提示

以上介绍的所有 transform 变换都可以进行 CoreAnimation 的动画展示。

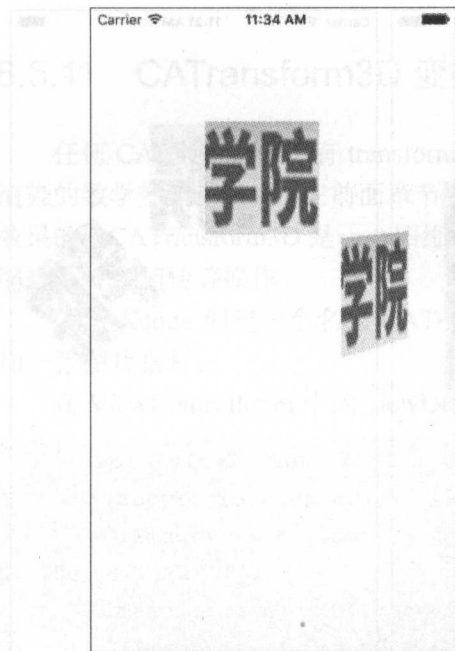


图 6-18 带景深效果的 transform 旋转变换

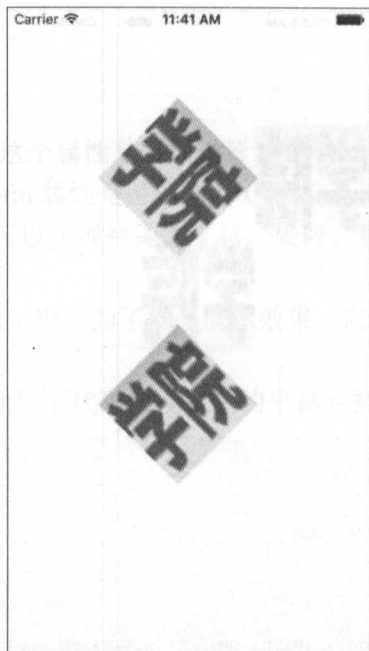


图 6-19 带翻转的 transform 旋转变换

6.6 炫酷的粒子效果

炫酷的动画效果是一款游戏必不可少的组成部分。前面向读者介绍了 CoreAnimation 框架中提供给开发者的一些动画方式，然而那些动画方式都是宏观上的、有序的、轨迹确定的规则的动画，对于微观的、无序的、轨迹无法预测的随机的动画，前边介绍的所有方法都无法满足需求，但是 CoreAnimation 框架不会令人失望，CAEmitterLayer 类可以创建出效果炫酷的粒子动画，有了这个类，将使创建一些十分复杂的动画效果变得容易而轻松。

6.6.1 粒子发射器——CAEmitterLayer

粒子效果的创建基于两个部分，一个部分是粒子发射器，用于配置粒子的整体效果，另一部分是粒子单元，用于配置粒子的具体属性。CAEmitterLayer 继承自 CALayer。

CAEmitterLayer 中常用的属性及意义如下所示。

//粒子单元数组，例如你在绘制火焰的效果时，你可以创建两个单元，一个单元负责烟雾，一个单元负责火苗。

```
@property(copy) NSArray *emitterCells;
```

//粒子的创建速率，默认为 1/s。

```
@property float birthRate;
```

//粒子的存活时间。默认为 1s。

```
@property float lifetime;
```

```

//发射器在 xy 平面的中心位置
@property CGPoint emitterPosition;
//发射器在 z 轴的位置
@property CGFloat emitterZPosition;
//发射器的尺寸大小
@property CGSize emitterSize;
//发射器的深度, 在某些模式下会产生立体效果
@property CGFloat emitterDepth;
//发射器的形状
@property(copy) NSString *emitterShape;
//发射器的发射模式
@property(copy) NSString *emitterMode;
//发射器渲染模式
@property(copy) NSString *renderMode;
//是否开启景深效果
@property BOOL preservesDepth;
//粒子的运动速度
@property float velocity;
//粒子的缩放大小
@property float scale;
//粒子的旋转位置
@property float spin;

```

上面属性中, **emitterShape** 属性决定粒子发射器的形状, 可选参数如下所示。

```

CA_EXTERN NSString * const kCAEmitterLayerPoint
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //点的形状, 粒子从一
个点发出
CA_EXTERN NSString * const kCAEmitterLayerLine
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //线的形状, 粒子从一
条线发出
CA_EXTERN NSString * const kCAEmitterLayerRectangle
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //矩形形状, 粒子从一
个矩形中发出
CA_EXTERN NSString * const kCAEmitterLayerCuboid
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //立方体形状, 会影响 z
平面的效果
CA_EXTERN NSString * const kCAEmitterLayerCircle
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //圆形, 粒子会在圆形
范围发射
CA_EXTERN NSString * const kCAEmitterLayerSphere
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //球型

```

emitterMode 属性决定粒子在粒子发射器中发射的位置, 可选参数如下所示。


```

CA_EXTERN NSString * const kCAEmitterLayerPoints
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //从发射器中发出
CA_EXTERN NSString * const kCAEmitterLayerOutline
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //从发射器边缘发出
CA_EXTERN NSString * const kCAEmitterLayerSurface
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //从发射器表面发出
CA_EXTERN NSString * const kCAEmitterLayerVolume
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //从发射器中点发出

```

renderMode 属性设置发射器的粒子渲染模式，可选参数如下所示。

```

CA_EXTERN NSString * const kCAEmitterLayerUnordered
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //这种模式下，粒子是
    无序出现的，多个发射源将混合
CA_EXTERN NSString * const kCAEmitterLayerOldestFirst
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //这种模式下，声明久
    的粒子会被渲染在最上层
CA_EXTERN NSString * const kCAEmitterLayerOldestLast
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //这种模式下，年轻的
    粒子会被渲染在最上层
CA_EXTERN NSString * const kCAEmitterLayerBackToFront
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //这种模式下，粒子的
    渲染按照 z 轴的前后顺序进行
CA_EXTERN NSString * const kCAEmitterLayerAdditive
    __OSX_AVAILABLE_STARTING (__MAC_10_6, __IPHONE_5_0); //这种模式会进行粒子混合

```

6.6.2 粒子单元——CAEmitterCell

CAEmitterLayer 中的属性主要对粒子的创建周期，发射速度等整体属性进行设置，而具体粒子的 UI 展现，颜色速度以及各种随机选项等是由粒子单元 CAEmitterCell 来进行设置与维护的。

CAEmitterCell 中常用属性及意义如下所示。

```

//类方法创建发射单元
+ (instancetype)emitterCell;
//设置发射单元的名称
@property(copy) NSString *name;
//粒子的创建速率
@property float birthRate;
//粒子的生存时间
@property float lifetime;
//粒子的生存时间容差
@property float lifetimeRange;
//粒子在 z 轴方向的发射角度

```

```

@property CGFloat emissionLatitude;
//粒子在 xy 平面的发射角度
@property CGFloat emissionLongitude;
//粒子发射角度的容差
@property CGFloat emissionRange;
//粒子的速度
@property CGFloat velocity;
//粒子速度的容差
@property CGFloat velocityRange;
//粒子 x, y, z 三个方向的加速度
@property CGFloat xAcceleration;
@property CGFloat yAcceleration;
@property CGFloat zAcceleration;
//粒子的缩放大小, 缩放容差和缩放速度
@property CGFloat scale;
@property CGFloat scaleRange;
@property CGFloat scaleSpeed;
//粒子的旋转度与旋转容差
@property CGFloat spin;
@property CGFloat spinRange;
//粒子的颜色
@property CGColorRef color;
//粒子在 R,G,B 三个色相上的容差和透明度的容差
@property float redRange;
@property float greenRange;
@property float blueRange;
@property float alphaRange;
//粒子在 RGB 三个色相上的变化速度和透明度的变化速度
@property float redSpeed;
@property float greenSpeed;
@property float blueSpeed;
@property float alphaSpeed;
//粒子的渲染对象, 可以设置为一个 CGImage 的对象
@property(strong) id contents;

```

6.6.3 创建粒子火焰动画

6.6.1 与 6.6.2 小节介绍了一些 iOS 中创建粒子效果相关类的基础知识, 本节将通过实战演练创建一个火焰效果的粒子动画来使读者更加深刻的理解粒子效果的创建方法, 更加熟练的应用粒子效果相关方法来完成复杂的粒子动画。

使用 Xcode 创建一个名为 MyFire 的工程, 在 ViewController.m 文件中声明如下属性。


```

@interface ViewController ()
{
    CAEmitterLayer * _fireEmitter;//发射器对象
}
@end

```

在 viewDidLoad 方法中添加如下代码。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    self.view.backgroundColor=[UIColor blackColor];
    //设置发射器
    _fireEmitter=[[CAEmitterLayer alloc] init];
    _fireEmitter.emitterPosition=CGPointMake(self.view.frame.size.width/2,
self.view.frame.size.height-20);
    _fireEmitter.emitterSize=CGSizeMake(self.view.frame.size.width-100, 20);
    _fireEmitter.renderMode = kCAEmitterLayerAdditive;
    //发射单元
    //火焰
    CAEmitterCell * fire = [CAEmitterCell emitterCell];
    fire.birthRate=1600;
    fire.lifetime=4.0;
    fire.lifetimeRange=1.5;
    fire.color=[[UIColor colorWithRed:0.8 green:0.4 blue:0.2 alpha:0.1]CGColor];
    fire.contents=(id)[[UIImage imageNamed:@"Particles_fire.png"]CGImage];
    [fire setName:@"fire"];
    fire.velocity=160;
    fire.velocityRange=80;
    fire.emissionLongitude=M_PI+M_PI_2;
    fire.emissionRange=M_PI_2;
    fire.scaleSpeed=0.3;
    fire.spin=0.2;

    //烟雾
    CAEmitterCell * smoke = [CAEmitterCell emitterCell];
    smoke.birthRate=800;
    smoke.lifetime=6.0;
    smoke.lifetimeRange=1.5;
    smoke.color=[[UIColor colorWithRed:1 green:1 blue:1 alpha:0.05]CGColor];
    smoke.contents=(id)[[UIImage imageNamed:@"Particles_fire.png"]CGImage];
    [smoke setName:@"smoke"];
    smoke.velocity=250;
    smoke.velocityRange=100;
}

```



```

smoke.emissionLongitude=M_PI+M_PI_2;
smoke.emissionRange=M_PI_2;
_fireEmitter.emitterCells=[NSArray arrayWithObjects:smoke,fire,nil];
[self.view.layer addSublayer:_fireEmitter];
}

```

上面代码中的 `Particles_fire` 图片只是一个白色的小色块，运行工程，效果如图 6-20 所示。

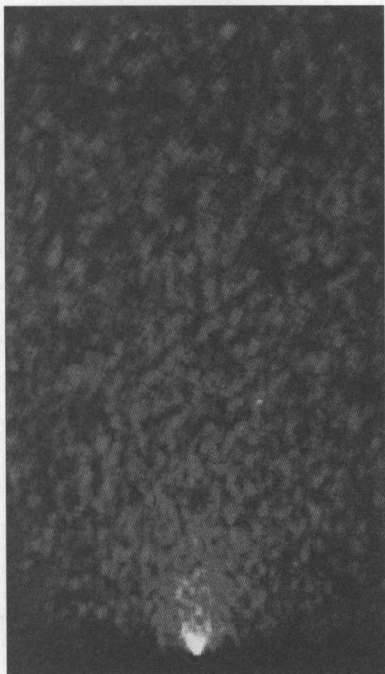


图 6-20 粒子火焰效果

6.7 播放 GIF 动态图

严格来讲，播放 GIF 动态图不能算是 iOS 中的动画开发技术，因为许多应用在进行某些需求的动画处理时，会使用播放 GIF 动态图的方式，所以本节也将这一技术归纳进 iOS 动画开发技术中。

一般有两种方式进行 GIF 格式图片的播放，一种是将其渲染为 `UIWebView` 进行播放，一种是将 GIF 文件中储存的图片数据与图片时序信息获取出来，使用 `UIImageView` 帧动画方式进行 GIF 动态图片的播放。

6.7.1 使用 `UIWebView` 进行 GIF 动态图播放

将 GIF 文件渲染为 `UIWebView` 进行动态图播放具有使用简单，效能强的特点，但也有一个缺点，就是在通过 `UIWebView` 进行加载 GIF 文件时，往往会耗费一定的时间。

使用 Xcode 创建一个名为 UIWebViewGIF 的工程, 把一个 GIF 动态图文件作为演示素材。在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIWebView * web = [[UIWebView alloc] initWithFrame:CGRectMake(20, 100,
280, 200)];
    //读取素材数据
    NSData * gifData = [NSData dataWithContentsOfFile:[NSBundle mainBundle]
pathForResource:@"gifImage" ofType:@"gif"]];
    [web loadData:gifData MIMEType:@"image/gif" textEncodingName:nil baseU
RL:nil];
    [self.view addSubview:web];
}
```

使用 UIWebView 加载 GIF 文件来播放 GIF 动态图这种方式播放的效果十分流畅, 而且开发者不需要对播放过程做任何管理, 全部由 UIWebView 来维护, 若非有特殊需求, 这种方式是开发中的首选。运行工程, 效果如图 6-21 所示。

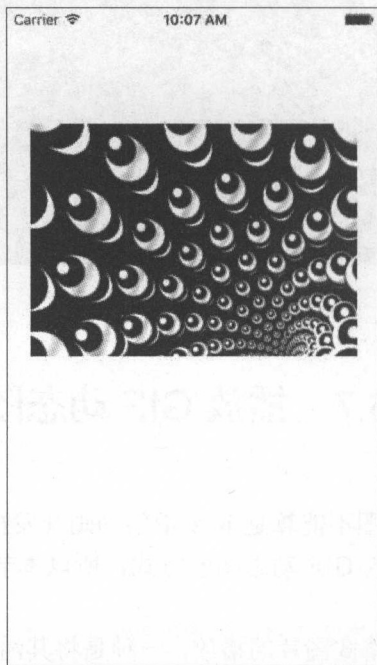


图 6-21 使用 UIWebView 播放 GIF 动态图

6.7.2 使用 UIImageView 帧动画进行 GIF 动态图播放

实际上, GIF 动态图文件中包含了一组图片及其信息数据, 这些信息数据中记录着这一组图片中各张图片的播放时长等信息, 开发者可以将图片和这些信息获取出来, 使用 UIImageView 的帧动画技术进行动画播放。

使用 Xcode 创建一个名为 UIImageViewGIF 的工程, 在 ViewController.m 文件中引入如下头文件用于处理 GIF 文件数据。

```
#import <ImageIO/ImageIO.h>
```

在 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIImageView * imageView = [[UIImageView alloc] initWithFrame:CGRectMake(
20, 100, 280, 200)];
    [self.view addSubview:imageView];
    NSString * dataPath = [[NSBundle mainBundle] pathForResource:@"gifImage
" ofType:@"gif"];
    //获取 GIF 文件数据
    CGImageSourceRef source = CGImageSourceCreateWithURL((CFURLRef) [NSURL
fileURLWithPath:dataPath], NULL);
    //获取 gif 文件中图片的个数
    size_t count = CGImageSourceGetCount(source);
    //定义一个变量记录 gif 播放一轮的时间
    float allTime=0;
    //存放所有图片
    NSMutableArray * imageArray = [[NSMutableArray alloc] init];
    //存放每一帧播放的时间
    NSMutableArray * timeArray = [[NSMutableArray alloc] init];
    //存放每张图片的宽度 (一般在一个 gif 文件中, 所有图片尺寸都会一样)
    NSMutableArray * widthArray = [[NSMutableArray alloc] init];
    //存放每张图片的高度
    NSMutableArray * heightArray = [[NSMutableArray alloc] init];
    //遍历
    for (size_t i=0; i<count; i++) {
        CGImageRef image = CGImageSourceCreateImageAtIndex(source, i, NULL);
        [imageArray addObject:(__bridge UIImage *) (image)];
        CGImageRelease(image);
        //获取图片信息
        NSDictionary * info = (__bridge NSDictionary*) CGImageSourceCopyPro
pertiesAtIndex(source, i, NULL);
        CGFloat width = [[info objectForKey:(__bridge NSString *) kCGImagePr
opertyPixelWidth] floatValue];
        CGFloat height = [[info objectForKey:(__bridge NSString *) kCGImageP
ropertyPixelHeight] floatValue];
        [widthArray addObject:[NSNumber numberWithFloat:width]];
        [heightArray addObject:[NSNumber numberWithFloat:height]];
    }
}
```



```

        NSDictionary * timeDic = [info objectForKey:(__bridge NSString *)kCGImagePropertyGIFDictionary];
        CGFloat time = [[timeDic objectForKey:(__bridge NSString *)kCGImagePropertyGIFDelayTime] floatValue];
        allTime+=time;
        [timeArray addObject:[NSNumber numberWithFloat:time]];
    }
    //添加帧动画
    CAKeyframeAnimation *animation = [CAKeyframeAnimation animationWithKeyPath:@"contents"];
    NSMutableArray * times = [[NSMutableArray alloc] init];
    float currentTime = 0;
    //设置每一帧的时间占比
    for (int i=0; i<imageArray.count; i++) {
        [times addObject:[NSNumber numberWithFloat:currentTime/allTime]];
        currentTime+=[timeArray[i] floatValue];
    }
    [animation setKeyTimes:times];
    [animation setValues:imageArray];
    [animation setTimingFunction:[CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionLinear]];
    //设置循环
    animation.repeatCount= MAXFLOAT;
    //设置播放总时长
    animation.duration = allTime;
    //Layer 层添加
    [[imageView layer] addAnimation:animation forKey:@"gifAnimation"];
}

```

可以发现，上面的方法十分繁琐，如果运行工程，可以看到这种方式播放的 GIF 动态图流畅性和性能也不如使用 UIWebView 的效果好，然而这种方式毕竟是使用 UIImageView 原生的方法来进行 GIF 动态播放的展示，在某些需求下十分易于扩展，读者可根据需求进行自由选择。

6.8 实战：小游戏 Flappy Bird 的设计与开发

综合本章所介绍内容，本节将与读者一起实现一款曾经风靡一时的小游戏 Flappy Bird。这款游戏也被称为像素小鸟。玩家通过单击屏幕来控制小鸟的飞行高度穿越高低不同的管道来获取分数。这款游戏因其操作简单、难度适当，使很多玩家既不需太多时间成本来学习，也可以在三两分钟的空闲时间中挑战几局，这是其成功的重要因素。

6.8.1 小鸟对象的设计

Objective-C 是一门面向对象的语言, 在 iOS 程序开发中也要求开发者使用面向对象的编程思想, 例如一个文本标签 UILabel 控件是一个对象, 一个图片视图 UIImageView 控件是一个对象, 通过对象间的组合与嵌套, 继承与扩展创建出更加复杂的对象, 由对象完成整个应用程序的界面与业务逻辑。在游戏开发中, 面向对象的思想更为重要, 在 Flappy Bird 游戏中, 首先应该设计游戏的主角: 像素鸟对象。

使用 Xcode 创建一个名为 Flappy Bird 的工程, 将游戏中所需要使用的素材添加进工程中, 读者可在本书提供的素材下载地址中找到所需素材。创建一个类取名为 Bird, 使其继承于 UIImageView。

在编写代码之前, 开发者应该思考要设计的对象具备哪些方法与属性, 以像素鸟对象为例, 它应该具有这样的功能或者属性: 扇动翅膀动作, 因重力而下落, 向上飞行等。在 Bird.h 文件中声明如下方法。

```
@interface Bird : UIImageView
//开始与结束飞行动作
-(void)startFlying;
-(void)stopFlying;
//开始与停止重力降落
-(void)startLand;
-(void)stopLand;
//向上飞
-(void)upFlay;
@end
```

在 Bird.m 文件中定义一个宏作为游戏世界的重力加速度常数。

```
#define G 4
```

在 Bird.m 文件中声明如下属性。

```
@implementation Bird
{
    //定时器
    NSTimer * _timer;
    //是否应该降落
    BOOL _couldLand;
    //下落速度
    float _speed;
}
@end
```

实现 Bird 类对象的初始化方法如下所示。

```

- (instancetype)init
{
    self = [super init];
    if (self) {
        //设置小鸟大小
        self.frame = CGRectMake(0, 0, 40, 31);
        self.image = [UIImage imageNamed:@"bird1"];
        //创建飞行动画
        NSMutableArray * array = [[NSMutableArray alloc] init];
        for (int i=0; i<3; i++) {
            UIImage * image = [UIImage imageNamed:[NSString stringWithFormat:@"bird%d", i+1]];
            [array addObject:image];
        }
        self.animationImages = array;
        self.animationDuration = 1;
        self.animationRepeatCount = 0;
        //初始化定时器与下落速度
        _timer = [NSTimer scheduledTimerWithTimeInterval:1/60.0 target:self
        selector:@selector(updateBird) userInfo:nil repeats:YES];
        _speed=0;
    }
    return self;
}

```

定时器的更新方法 `updateBird` 方法实现如下所示。

```

-(void)updateBird{
    if (!_couldLand) {
        //y 坐标增大 小鸟掉落
        self.center = CGPointMake(self.center.x, self.center.y+_speed);
        _speed +=G/60.0;
        //超出边界做处理
        if (self.center.y<=0) {
            self.center = CGPointMake(self.center.x, 0);
        }else if (self.center.y+self.frame.size.height/2>=self.superview.frame.size.height){
            self.center = CGPointMake(self.center.x, self.superview.frame.size.height-self.frame.size.height/2);
        }
        }else{
            _speed=0;
        }
    }
}

```


在上面的代码中做了一些边界处理，保证小鸟的坐标始终控制在屏幕内，实现 Bird.h 中声明的相应方法如下所示。

```
-(void)startFlying{
    if (self.isAnimating) {
        return;
    }
    [self startAnimating];
}
-(void)stopFlying{
    if (self.isAnimating) {
        [self stopAnimating];
    }
}
-(void)startLand{
    _couldLand = YES;
}
-(void)stopLand{
    _couldLand = NO;
}
-(void)upFlay{
    //给小鸟一个向上的初速度
    _speed = -2.5;
}
```

通过上面的代码，基本完成了游戏主角小鸟类的设计，读者可在 ViewController.m 文件中创建对象进行功能的测试。

6.8.2 游戏开始界面的设计

任何游戏不可能一打开就立马开始游戏，都会有一个等待开始的界面，等玩家单击某个开始按钮来触发游戏的开始。Flayyp Bird 游戏也需要这样一个界面。使用 Xcode 新创建一个类，取名为 GameStartView，使其继承于 UIView。在 GameStartView 中声明如下属性和方法并定义一个协议。

```
@protocol GameStartViewDelegate<NSObject>
-(void)gameStartViewTouchStart;
@end
@interface GameStartView : UIView
@property(n nonatomic, weak) id<GameStartViewDelegate> delegate;
-(void)show;
-(void)unshow;
@end
```

因为开始游戏的界面总会内嵌一个开始按钮，GameStartViewDelegate 协议用于按钮触发方法的回调。在 GameStartView.m 中实现初始化方法如下所示。

```
- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        self.backgroundColor = [UIColor clearColor];
        UIImageView * bg = [[UIImageView alloc] initWithFrame:CGRectMake(se
lf.frame.size.width/2-128, 100, 256, 73)];
        bg.image = [UIImage imageNamed:@"main"];
        [self addSubview:bg];
        //开始按钮
        UIButton * btn = [UIButton buttonWithTypeCustom];
        btn.frame = CGRectMake(self.frame.size.width/2-67, 350, 135, 75);
        [btn setBackgroundImage:[UIImage imageNamed:@" start"] forState:UI
ControlStateNormal];
        [btn addTarget:self action:@selector(touch) forControlEvents:UICon
trolEventTouchUpInside];
        [self addSubview:btn];
    }
    return self;
}
```

按钮的触发方法 touch 实现如下所示。

```
-(void)touch{
    [self unshow];
    if ([self.delegate respondsToSelector:@selector(gameStartViewTouchSta
rt)]) {
        [self.delegate gameStartViewTouchStart];
    }
}
```

touch 方法中使用了 respondsToSelector:这样一个方法，这个方法用于检测某个对象的类中是否实现了某一方法。实现用于展示和隐藏界面的 show 和 unshow 方法如下所示。

```
-(void)show{
    [UIView animateWithDuration:0.3 animations:^(
        self.alpha=1;
    )];
}
-(void)unshow{
    [UIView animateWithDuration:0.3 animations:^(
```

```

        self.alpha=0;
    }];
}

```

6.8.3 游戏结束界面的设计

Flappy Bird 游戏中有这样一条规则：当小鸟落地或者碰撞到管道时，游戏即会结束。因此开发者还需要创建一个用于游戏结束后展现玩家所得分数的界面。在工程中创建一个新的类 `GameOverView`，使其继承于 `UIView`。在 `GameOverView.h` 文件中声明如下属性和方法并定义一个协议。

```

@protocol GameOverViewDelegate<NSObject>
-(void)gameOverViewUnShow;
@end
@interface GameOverView : UIView
-(void)show;
-(void)unshow;
-(void)setSource:(int)source;
@property (nonatomic, weak) id<GameOverViewDelegate>delegate;
@end

```

`GameOverViewDelegate` 用于提供游戏结束界面将消失时的回调，`GameOverView` 类的 `setSource:` 方法用于设置玩家在游戏中获取的分数。

在 `GameOverView.m` 文件中声明一个 `UILabel` 标签控件作为计分板，如下所示。

```

@implementation GameOverView
{
    //计分板
    UILabel * _label;
}
@end

```

实现 `GameOverView` 类的初始化方法，如下所示。

```

- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        self.backgroundColor = [UIColor clearColor];
        UIImageView * bg = [[UIImageView alloc] initWithFrame:CGRectMake(self.frame.size.width/2-124, 100, 248, 56)];
        bg.image = [UIImage imageNamed:@"gameover2"];
        [self addSubview:bg];
        _label = [[UILabel alloc] initWithFrame:CGRectMake(self.frame.size.width/2-25, 200, 50, 50)];
    }
}

```



```

        _label.backgroundColor = [UIColor clearColor];
        _label.textAlignment = NSTextAlignmentCenter;
        _label.font = [UIFont systemFontOfSize:23];
        _label.textColor = [UIColor redColor];
        [self addSubview:_label];
        self.alpha = 0;
    }
    return self;
}

```

开发者还需要在游戏结束的界面实现这样的需求:当用户单击屏幕后,结束游戏界面逐渐消失,游戏开始界面重新展现。因此,在 `GameOverView.m` 中实现如下方法。

```

-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{
    [self unshow];
    if ([self.delegate respondsToSelector:@selector(gameOverViewUnShow)]) {
        [self.delegate gameOverViewUnShow];
    }
}

```

在 `GameOverView.m` 中实现 `GameOverView.h` 中声明的方法,如下所示。

```

-(void)show{
    [UIView animateWithDuration:0.3 animations:^(
        self.alpha = 1;
    )];
}
-(void)unshow{
    [UIView animateWithDuration:0.3 animations:^(
        self.alpha = 0;
    )];
}
-(void)setSource:(int)source{
    _label.text = [NSString stringWithFormat:@"%d",source];
}

```

6.8.4 Flappy Bird 游戏主框架的搭建

通过前面小节的准备工作,已经将游戏中需要使用的独立控件进行了设计,开发者还需搭建游戏的主功能界面并将前面的独立控件进行组合与运用。

在 `ViewController.m` 文件中引入如下头文件。

```

#import "Bird.h"
#import "GameStartView.h"
#import "GameOverView.h"

```

在 ViewController.m 中声明如下的属性并遵守相应的协议。

```
@interface ViewController ()<GameStartViewDelegate,GameOverViewDelegate>
{
    //游戏主角像素鸟
    Bird * _bird;
    //背景
    UIImageView * _bg1;
    UIImageView * _bg2;
    //定时器
    NSTimer * _timer;
    //第一对管道
    UIImageView * _woodUp1;
    UIImageView * _woodDown1;
    //第二对管道
    UIImageView * _woodUp2;
    UIImageView * _woodDown2;
    //地面
    UIImageView * _floor1;
    UIImageView * _floor2;
    //游戏开始于结束视图
    GameStartView * _startView;
    GameOverView * _overView;
    //分数统计
    UILabel * _sourceLabel;
    int _source;
}
@end
```

在 ViewController.m 文件的 viewDidLoad 方法中编写如下初始化代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    //初始化分数
    _source=0;
    //创建背景
    [self creatBG];
    //创建管道
    [self creatWood];
    //创建地板
    [self creatFloor];
    //初始化像素鸟
    _bird = [[Bird alloc]init];
    //设置位置
```



```

    _bird.center = CGPointMake(100, 300);
    [self.view addSubview:_bird];
    //开始飞行动画
    [_bird startFlying];
    //初始化游戏开始于结束视图
    _startView = [[GameStartView alloc] initWithFrame:self.view.frame];
    _overView = [[GameOverView alloc] initWithFrame:self.view.frame];
    [self.view addSubview:_startView];
    [self.view addSubview:_overView];
    _startView.delegate=self;
    _overView.delegate=self;
    //初始化定时器
    _timer = [NSTimer scheduledTimerWithTimeInterval:1/60.0 target:self selector:@selector(updateUI) userInfo:nil repeats:YES];
    //将定时器暂停
    _timer.fireDate = [NSDate distantFuture];
    //创建计分板
    [self creatSourceLabel];
}

```

创建背景的 creatBG 方法实现，如下所示。

```

-(void)creatBG{
    _bg1 = [[UIImageView alloc] initWithFrame:self.view.bounds];
    _bg1.image = [UIImage imageNamed:@"bg"];
    _bg2 = [[UIImageView alloc] initWithFrame:CGRectMake(self.view.frame.size.width, 0, self.view.frame.size.width, self.view.frame.size.height)];
    _bg2.image = [UIImage imageNamed:@"bg"];
    [self.view addSubview:_bg1];
    [self.view addSubview:_bg2];
}

```

创建管道的方法 creatWood 实现，如下所示。

```

-(void)creatWood{
    //一屏最多同时出现 2 组管道
    _woodUp1 = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"04
    "]];
    _woodDown1 = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"0
    5"]];
    _woodUp2 = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"04
    "]];
    _woodDown2 = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"0
    5"]];
    //取一个随机数作为初始高度
}

```



```

float sH = self.view.frame.size.height;
//下面柱子高度最少为 150
//柱子间缝隙为 100
//上面柱子的高度最少为 150
int parm = sH-150-100-150;
int height = arc4random()%parm+150;
_woodUp1.frame = CGRectMake(600, height-325, 54, 325);
_woodDown1.frame = CGRectMake(600, height+100, 54, 325);
height = arc4random()%parm;
_woodUp2.frame = CGRectMake(780, height-325, 54, 325);
_woodDown2.frame = CGRectMake(780, height+100, 54, 325);
[self.view addSubview:_woodUp1];
[self.view addSubview:_woodUp2];
[self.view addSubview:_woodDown1];
[self.view addSubview:_woodDown2];
}

```

创建地板的 `creatFloor` 方法实现，如下所示。

```

-(void)creatFloor{
    _floor1 = [[UIImageView alloc] initWithFrame:CGRectMake(0, self.view.frame.size.height-112, 336, 112)];
    _floor2 = [[UIImageView alloc] initWithFrame:CGRectMake(self.view.frame.size.width, self.view.frame.size.height-112, 336, 112)];
    _floor1.image = [UIImage imageNamed:@"03"];
    _floor2.image = [UIImage imageNamed:@"03"];
    [self.view addSubview:_floor1];
    [self.view addSubview:_floor2];
}

```

定时器的触发方法 `updateUI` 主要有下面几个作用：

- 控制背景的移动及进行背景复用
- 控制管道的移动及进行管道复用
- 控制地板的移动及进行地板复用
- 进行分数的更新
- 进行游戏是否结束的判定

根据如上需求，方法实现如下所示。

```

-(void)updateUI{
    float x = _bg1.frame.origin.x;
    if (x<=-self.view.frame.size.width) {
        _bg1.frame=self.view.bounds;
        _bg2.frame = CGRectMake(self.view.frame.size.width, 0, self.view.frame.size.width, self.view.frame.size.height);
    }
}

```

```

        _floor1.frame = CGRectMake(0, self.view.frame.size.height-112, 336,
112);
        _floor2.frame = CGRectMake(self.view.frame.size.width, self.view.frame.size.height-112, 336, 112);
    }
    _bg1.frame = CGRectMake(_bg1.frame.origin.x-1, 0, self.view.frame.size.width, self.view.frame.size.height);
    _bg2.frame = CGRectMake(_bg2.frame.origin.x-1, 0, self.view.frame.size.width, self.view.frame.size.height);
    _floor1.frame = CGRectMake(_floor1.frame.origin.x-1, self.view.frame.size.height-112, 336, 112);
    _floor2.frame = CGRectMake(_floor2.frame.origin.x-1, self.view.frame.size.height-112, 336, 112);
    _woodUp1.frame = CGRectMake(_woodUp1.frame.origin.x-1, _woodUp1.frame.origin.y, _woodUp1.frame.size.width, _woodUp1.frame.size.height);
    _woodUp2.frame = CGRectMake(_woodUp2.frame.origin.x-1, _woodUp2.frame.origin.y, _woodUp2.frame.size.width, _woodUp2.frame.size.height);
    _woodDown1.frame = CGRectMake(_woodDown1.frame.origin.x-1, _woodDown1.frame.origin.y, _woodDown1.frame.size.width, _woodDown1.frame.size.height);
    _woodDown2.frame = CGRectMake(_woodDown2.frame.origin.x-1, _woodDown2.frame.origin.y, _woodDown2.frame.size.width, _woodDown2.frame.size.height);
    if (_woodUp1.frame.origin.x+_woodUp1.frame.size.width<=0) {
        //将其往后放
        float sH = self.view.frame.size.height;
        int parm = sH-150-100-150;
        int height = arc4random()%parm+150;
        _woodUp1.frame = CGRectMake(_woodUp2.frame.origin.x+280, height-_woodUp2.frame.size.height, _woodUp2.frame.size.width, _woodUp2.frame.size.height);
        _woodDown1.frame = CGRectMake(_woodUp2.frame.origin.x+280, height+100, _woodUp2.frame.size.width, _woodUp2.frame.size.height);
    }
    if (_woodUp2.frame.origin.x+_woodUp2.frame.size.width<=0) {
        //将其往后放
        float sH = self.view.frame.size.height;
        int parm = sH-150-100-150;
        int height = arc4random()%parm+150;
        _woodUp2.frame = CGRectMake(_woodUp1.frame.origin.x+280, height-_woodUp1.frame.size.height, _woodUp1.frame.size.width, _woodUp1.frame.size.height);
        _woodDown2.frame = CGRectMake(_woodUp1.frame.origin.x+280, height+100, _woodUp1.frame.size.width, _woodUp1.frame.size.height);
    }
}

```



```

//进行分数更新
if (_bird.frame.origin.x==_woodUp1.frame.origin.x+_woodUp1.frame.size.
width) {
    _source+=1;
    _sourceLabel.text = [NSString stringWithFormat:@"%d",_source];
}
//进行死亡判定
[self ifDead];
}

```

死亡判定的方法 ifDead 的实现方法如下所示。

```

-(void)ifDead{
    //落地
    if (_bird.frame.origin.y+_bird.frame.size.height>_floor1.frame.origin.
y) {
        //死亡
        [_bird stopFlying];
        [_bird stopLand];
        _timer.fireDate = [NSDate distantFuture];
        [_overview setSource:_source];
        [_overview show];
        _sourceLabel.hidden=YES;
        _source=0;
    }
    //碰上管道
    if (CGRectIntersectsRect(_bird.frame, _woodUp1.frame)||CGRectIntersec
tsRect(_bird.frame, _woodUp2.frame)||CGRectIntersectsRect(_bird.frame, _wood
Down1.frame)||CGRectIntersectsRect(_bird.frame, _woodDown2.frame)) {
        //死亡
        [_bird stopFlying];
        [_bird stopLand];
        _timer.fireDate = [NSDate distantFuture];
        [_overview setSource:_source];
        [_overview show];
        _sourceLabel.hidden=YES;
        _source=0;
    }
}
}

```

创建计分板 creatSourceLabel 的方法实现如下所示。

```

-(void)creatSourceLabel{
    _sourceLabel = [[UILabel alloc] initWithFrame:CGRectMake(self.view.fra
me.size.width/2-25, 100, 50, 50)];
}

```



```

        _sourceLabel.backgroundColor = [UIColor clearColor];
        _sourceLabel.textAlignment = NSTextAlignmentCenter;
        _sourceLabel.font = [UIFont systemFontOfSize:23];
        _sourceLabel.textColor = [UIColor redColor];
        _sourceLabel.hidden=YES;
        [self.view addSubview:_sourceLabel];
    }

```

实现开始游戏与游戏结束界面的代理方法如下所示。

```

-(void)gameOverViewUnShow{
    [_startView show];
    //进行初始化设置
    _bird.center = CGPointMake(100, 300);
    [_bird startFlying];
    float sH = self.view.frame.size.height;
    //下面柱子高度最少为 150
    //柱子间缝隙为 100
    //上面柱子的高度最少为 150
    int parm = sH-150-100-150;
    int height = arc4random()%parm+150;
    _woodUp1.frame = CGRectMake(600, height-325, 54, 325);
    _woodDown1.frame = CGRectMake(600, height+100, 54, 325);
    height = arc4random()%parm;
    _woodUp2.frame = CGRectMake(780, height-325, 54, 325);
    _woodDown2.frame = CGRectMake(780, height+100, 54, 325);

}

-(void)gameStartViewTouchStart{
    [_startView unshow];
    _timer.fireDate = [NSDate distantPast];
    _sourceLabel.hidden=NO;
    [_bird startLand];
}

```

最后还需要实现一个方法，就是当玩家单击屏幕的时候，像素小鸟会向上飞行一定高度。

```

-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{
    [_bird upFly];
}

```

通过上面代码的编写，一款简易的 Flappy Bird 游戏就开发完成了，运行工程，其主要界面效果如图 6-22~图 6-24 所示。

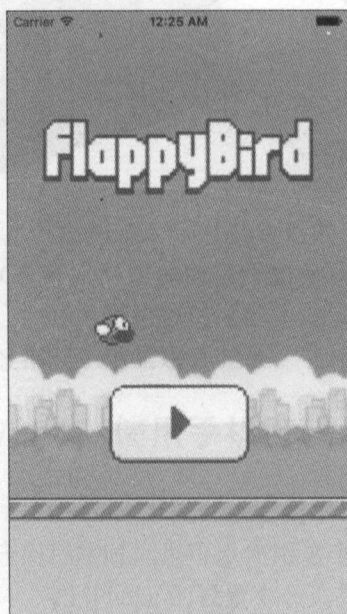


图 6-22 开始游戏界面

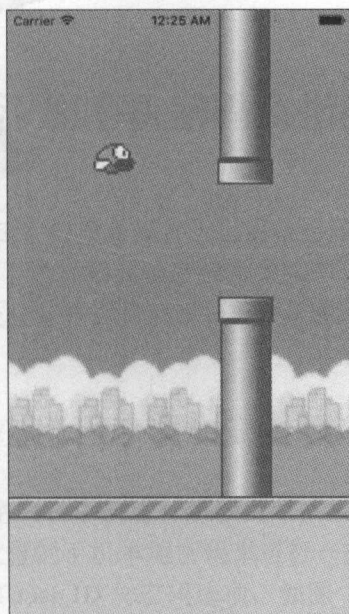


图 6-23 游戏中界面

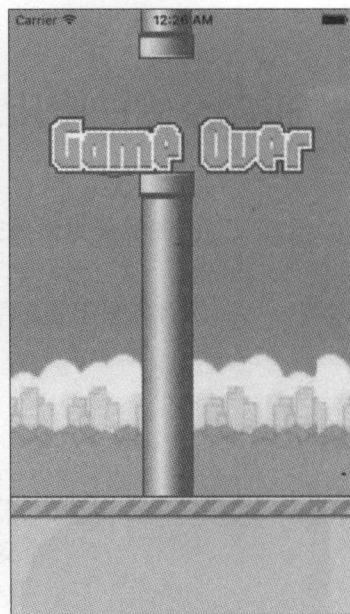


图 6-24 游戏结束界面

第 7 章

传感器开发

如果将智能手机中的 CPU 芯片、GPU 芯片、内存等元件比喻成手机的大脑和心脏,那么手机中的各种传感器芯片就是其接触外界的感官系统。各种传感器的协作完成智能手机的各种智能功能,例如行车定位中的速度与加速度,横竖屏切换时的手机方向定位,根据光线强弱进行屏幕亮度的调节等。本章将介绍在 iOS 应用开发中一些常用的传感器接口,其中包括 iPhone 5s 之后新加入的指纹识别传感器、加速度传感器、螺旋仪传感器、磁力传感器、距离传感器与蓝牙传感器等。

通过本章的学习,读者能够掌握:

1. 在 iPhone 5s 及以上设备使用指纹传感器进行安全验证。
2. 综合使用加速度传感器、螺旋仪传感器和磁力传感器获取设备在空间中的形态。
3. 使用距离传感器获取手机前方遮挡状态。
4. 使用蓝牙传感器进行设备间通信。
5. 在应用程序中嵌入地图模块。
6. 使用 iOS 原生地图框架进行导航和附近检索等功能。
7. 实战开发一款蓝牙对战的五子棋游戏。

7.1 为应用程序添加手机密码及指纹识别的安全验证

在 iPhone 5s 及其之后的 iOS 手机设备硬件上已经支持了进行指纹验证的功能，相应的在 iOS 开发框架中也为开发者添加了进行指纹验证的相关接口。

开发者可以使用的本地安全验证方式有两种，一种是通过手机密码来进行验证，一种是通过指纹识别来进行验证。

7.1.1 使用手机密码为应用程序添加安全验证

iPhone 手机可以设置本地密码在手机进行解锁的时候验证密码保护用户的隐私。在开发应用程序时，也可以使用这个设置的手机本地密码来进行一些安全验证。

在 iPhone 的设置中，选择 Touch ID 与密码选项，如图 7-1 所示。

如果此 iPhone 手机中还没有设置过密码，则在弹出的设置界面中选择打开密码，如图 7-2 所示。

设置了手机密码后，在手机进行解锁时，会提示用户输入手机密码，如图 7-3 所示。



图 7-1 系统设置界面

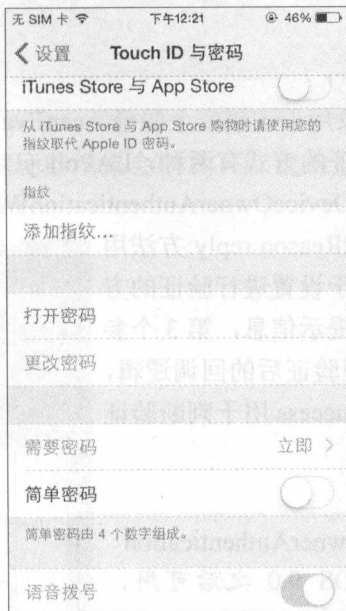


图 7-2 进行手机密码设置



图 7-3 iPhone 解锁界面

出现如图 7-3 所示的解锁界面，说明 iPhone 手机密码已经设置成功，下面在应用开发中添加密码验证的功能。

使用 Xcode 创建一个名为 AuthenticationTestOne 的工程，在其 ViewController.m 文件中添加如下头文件。

```
#import <LocalAuthentication/LocalAuthentication.h>
```

在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    //创建安全验证对象
    LAContext * con = [[LAContext alloc] init];
    NSError * error;
    //判断是否支持密码验证
    /**
     *LAPolicyDeviceOwnerAuthentication 手机密码的验证方式
     *LAPolicyDeviceOwnerAuthenticationWithBiometrics 指纹的验证方式
     */
    BOOL can = [con canEvaluatePolicy:LAPolicyDeviceOwnerAuthentication error:&error];
    if (can) {
        [con evaluatePolicy:LAPolicyDeviceOwnerAuthentication localizedReason:@"请输入您的手机密码进行安全验证" reply:^(BOOL success, NSError * _Nullable error) {
            NSLog(@"%d,%@", success, error);
        }];
    }
}
```

上面代码中，LAContext 对象用于连接安全验证，canEvaluatePolicy:error:方法用于进行是否支持安全验证的判断，支持验证的方式有两种，LAPolicyDeviceOwnerAuthentication 是使用手机密码的验证方法，LAPolicyDeviceOwnerAuthenticationWithBiometrics 是使用用户指纹的验证方式。evaluatePolicy:localizedReason:reply:方法用于进行验证，其中第 1 个参数用于设置进行验证的方式，第 2 个参数用于设置验证的提示信息，第 3 个参数是一个 block 代码块，用于处理验证后的回调逻辑，block 中返回的 BOOL 类型参数 success 用于判断验证是否成功。



提示

LAPolicyDeviceOwnerAuthentication 方式的验证在 iOS 9.0 之后可用，LAPolicyDeviceOwnerAuthenticationWithBiometrics 的验证方式在 iOS 8.0 之后可用。

运行工程，进行密码验证的界面效果如图 7-4 所示。

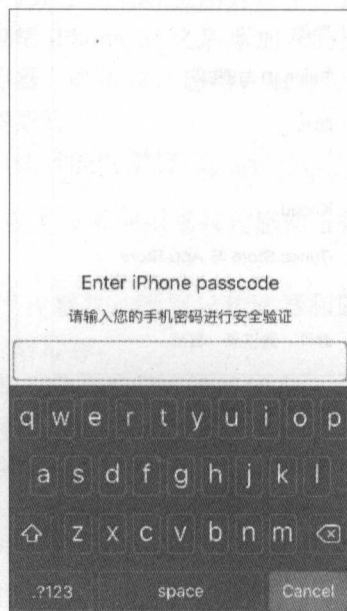


图 7-4 在应用中添加手机密码验证的功能

7.1.2 使用用户指纹为应用程序添加安全验证

为应用程序添加指纹识别验证的前提是用户设置开启的指纹验证功能。使用 Xcode 创建一个名为 AuthenticationTestTwo 的工程，在 ViewController.m 文件中添加如下头文件。

```
#import <LocalAuthentication/LocalAuthentication.h>
```

在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    LAContext * con = [[LAContext alloc] init];
    NSError * error;
    BOOL can = [con canEvaluatePolicy:LAPolicyDeviceOwnerAuthenticationWithBiometrics error:&error];
    NSLog(@"%d", can);
    if (can) {
        [con evaluatePolicy:LAPolicyDeviceOwnerAuthenticationWithBiometrics localizedReason:@"验证指纹" reply:^(BOOL success, NSError * _Nullable error) {
            NSLog(@"%d,%@", success, error);
        }];
    }
}
```

与进行收集密码验证相似，验证的结果会以 BOOL 值的形式传递进 block 代码块中。指纹验证界面效果如图 7-5 所示。



提示

关于指纹验证的测试只可以在真机上进行验证。

图 7-5 进行指纹验证的界面

7.2 使用加速度传感器、螺旋仪传感器与磁力传感器获取设备空间状态

加速度传感器又被称为重力加速度传感器，其与螺旋仪传感器、磁力传感器综合使用可以精确地定位出设备的空间形态，例如朝向和旋转角度等。在 iOS 开发中，对于这些传感器的应用有两个平行的框架，在 iOS 5 之前的 iOS 设备所支持的传感器有限，只有重力加速度传感器，关于设备的重力加速度信息由 `UIAccelerometer` 这个类负责，在 iOS 5 之后，随着 iPhone 设备所支持的传感器的完善，有关设备空间位置的相关信息交由 `CoreMotion` 框架负责，`CoreMotion` 框架中封装了各个传感器协同合作的数据以及一些转换算法，比之前的 `UIAccelerometer` 强大许多。

7.2.1 使用 `UIAccelerometer` 获取设备空间状态

`UIAccelerometer` 类采用单例的设计模式，通过代理方法实时地将设备的重力加速度信息返回给开发者。使用 Xcode 创建一个名为 `UIAccelerometerTest` 的工程，在 `ViewController.m` 文件中遵守如下协议。

```
@interface ViewController ()<UIAccelerometerDelegate>
@end
```

在 `ViewController.m` 文件的 `viewDidLoad` 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    //设置代理
    [UIAccelerometer sharedAccelerometer].delegate = self;
    //设置更新频率
    [UIAccelerometer sharedAccelerometer].updateInterval = 1;
}
```

在 `ViewController.m` 文件中实现如下代理方法。

```
-(void)accelerometer:(UIAccelerometer *)accelerometer didAccelerate:(UIAcceleration *)acceleration{
    NSLog(@"\n%f\n%f\n%f", acceleration.x, acceleration.y, acceleration.z);
    NSLog(@"%f", acceleration.timestamp);
}
```

`accelerometer:didAccelerate:`方法是加速度传感器的检测回调方法，在这个方法中会传入一个 `UIAcceleration` 类型的参数，这个参数会将设备在空间中的 x 轴加速度值、y 轴加速度值和 z 轴加速度值传递进来，`UIAcceleration` 中的属性如下所示。

```
//加速度传感器的时间戳
@property(nonatomic,readonly) NSTimeInterval timestamp;
//x 轴方向的加速度分量
@property(nonatomic,readonly) UIAccelerationValue x;
//y 轴方向的加速度分量
@property(nonatomic,readonly) UIAccelerationValue y;
//z 轴方向的加速度分量
@property(nonatomic,readonly) UIAccelerationValue z;
```



提示

由于 UIAccelerometer 在 iOS 5 系统之后已经被弃用,在项目中使用 UIAccelerometer 类及其相关方法会被 Xcode 报警,读者不必在意。

7.2.2 使用 CoreMotion 框架获取设备空间状态信息

CoreMotion 框架相比 UIAcceleration 类强大很多,它不仅将加速度传感器、螺旋仪传感器和磁力传感器进行统一的配置和管理,其中还封装了许多转换算法,可以帮助开发者直接获取设备的运动信息。

CoreMotion 框架主要负责处理 4 种数据,一种是加速度数据,一种是螺旋仪数据,一种是磁感应数据,最后一种是前 3 种数据通过复杂运算得到的设备的运动信息数据。

设备的加速度信息封装在 CMAccelerometerData 类中,这个类中有个 CMAcceleration 类型的结构体,这个结构体中存放着具体的加速度信息,如下所示。

```
typedef struct {
    double x;
    double y;
    double z;
} CMAcceleration;
```

设备的螺旋仪信息封装在 CMGyroData 类中,这个类中有一个 CMRotationRate 类型的结构体,这个结构体中存放着设备的螺旋仪角度信息,如下所示。

```
typedef struct {
    double x;
    double y;
    double z;
} CMRotationRate;
```

设备的磁感应信息封装在 CMMagnetometerData 类中,这个类中有一个 CMMagneticField 类型的结构体,这个结构体中存放着设备各个方向的磁感应信息,如下所示。

```
typedef struct {
    double x;
    double y;
```

```
double z;
} CMMagneticField;
```

相比于前面 3 种设备信息，第 4 种设备的运动信息是前面 3 种的结合与推导，比前面 3 种数据也要复杂一些，由 `CMDeviceMotion` 类定义，这个类中定义了如下属性。

```
//设备的状态对象
@property(readonly, nonatomic) CMAAttitude *attitude;
//设备的角速度
@property(readonly, nonatomic) CMRotationRate rotationRate;
//设备的重力加速度
@property(readonly, nonatomic) CMAcceleration gravity;
//用户加给设备的加速度 设备的总加速度为重力加速度加上用户给的加速度
@property(readonly, nonatomic) CMAcceleration userAcceleration;
//设备的磁场矢量对象
@property(readonly, nonatomic) CMCalibratedMagneticField magneticField;
```

在上面列出的属性中，设备的状态信息对象 `attitude` 中又封装了一些设备的基础属性，`CMAAttitude` 类中属性列举如下所示。

```
//设备的欧拉角 roll
@property(readonly, nonatomic) double roll;
//设备的欧拉角 pitch
@property(readonly, nonatomic) double pitch;
//设备的欧拉角 yaw
@property(readonly, nonatomic) double yaw;
//设备状态的旋转矩阵
@property(readonly, nonatomic) CMRotationMatrix rotationMatrix;
//设备状态的四元数
@property(readonly, nonatomic) CMQuaternion quaternion;
```

使用 Xcode 创建一个名为 `CoreMotionTest` 的工程，在工程中使用 `CoreMotion` 框架监听设备信息有两种方式。

1. 使用定时器监听的方式主动拉取设备信息

在 `ViewController.m` 文件中引入如下头文件。

```
#import <CoreMotion/CoreMotion.h>
```

在 `ViewController.m` 文件中声明如下属性。

```
@interface ViewController ()
{
    CMMotionManager * manager;
}
@end
```


在 `viewDidLoad` 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    //创建管理对象
    manager= [[CMMotionManager alloc] init];
    //开启加速度更新
    [manager startAccelerometerUpdates];
    //开启螺旋仪更新
    [manager startGyroUpdates];
    //开启状态更新
    [manager startMagnetometerUpdates];
    //创建定时器
    NSTimer * time = [NSTimer scheduledTimerWithTimeInterval:1 target:self
selector:@selector(update) userInfo:nil repeats:YES];
    time.fireDate = [NSDate distantPast];
}
```

在上面的代码中, `CMMotionManager` 类对象调用 `startAccelerometerUpdates` 方法开始进行设备加速度信息的获取, 调用 `startGyroUpdates` 方法开始进行设备螺旋仪信息的获取, 调用 `startMagnetometerUpdates` 方法开始进行设备运动状态信息的获取。实现定时器刷新的方法如下所示。

```
- (void)update{
    //获取数据
    NSLog(@"%f,%f,%f\n%f,%f,%f",manager.accelerometerData.acceleration.x,manager.accelerometerData.acceleration.y,manager.accelerometerData.acceleration.z,manager.gyroData.rotationRate.x,manager.gyroData.rotationRate.y,manager.gyroData.rotationRate.z);
}
```

2. 使用 block 回调的方式获取设备信息

将 `ViewController.m` 文件中 `viewDidLoad` 方法中的代码修改如下所示。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    //创建管理对象
    manager= [[CMMotionManager alloc] init];
    //在当前线程中回调
    [manager startAccelerometerUpdatesToQueue:[NSOperationQueue currentQueue]
withHandler:^(CMAccelerometerData * _Nullable accelerometerData, NSError * _Nullable error) {
```

```

    NSLog(@"%f,%f,%f\n%f,%f,%f",manager.accelerometerData.acceleration.x,manager.accelerometerData.acceleration.y,manager.accelerometerData.acceleration.z,manager.gyroData.rotationRate.x,manager.gyroData.rotationRate.y,manager.gyroData.rotationRate.z);
  }
}

```

CMMotionManager 类对象的 startAccelerometerUpdatesToQueue:withHandler: 方法用于开始获取设备的加速度数据，并在回调 block 中将设备加速度相关信息传递给开发者。与上面方法对应，下面的方法用于开启获取设备螺旋仪数据，磁力传感器数据与设备运动数据。

```

//开始获取螺旋仪数据
- (void)startGyroUpdatesToQueue:(NSOperationQueue *)queue withHandler:(CMGyroHandler)handler;
//开始获取磁力传感器
- (void)startMagnetometerUpdatesToQueue:(NSOperationQueue *)queue withHandler:(CMMagnetometerHandler)handler;
//开始获取设备运动信息
- (void)startDeviceMotionUpdatesToQueue:(NSOperationQueue *)queue withHandler:(CMDeviceMotionHandler)handler;

```

上面两种方式在写法上第二种使用 block 回调的方式更加简单，并且第二种采用 push 的方式获取信息，性能上也比第一种要好，所以在开发中，读者应尽量采用第二种方式。

7.3 距离传感器的应用

iPhone 手机中内置了距离传感器，其位置在手机的听筒附近，当有物体靠近手机的听筒附近时，其会进行感应触发。iPhone 手机的距离感应器在生活中最广泛的应用就在于当用户在进行手机通话时，手机屏幕一靠近脸颊屏幕就会自动关闭。

在 iOS 开发中，系统并没有暴露给开发者关于距离感应器的过多接口，开发者只通过 UIDevice 类中的一个开关属性来监听距离感应器的状态。

使用 Xcode 创建一个名为 ProximityMonitorTest 的工程，在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    //开启获取设备距离感应器数据
    [UIDevice currentDevice].proximityMonitoringEnabled = YES;
    //添加监听距离感应器数据的变化
    [[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(notice) name:UIDeviceProximityStateDidChangeNotification object:nil];
}

```

UIDevice 类是一个单例类，其用于获取当前设备的一些设备信息，例如设备名称、设备系统版本号和设备类型等，其也可以监听设备的电池状态信息，设备方向信息和内置距离传感器的信息等。使用 `currentDevice` 方法获取 UIDevice 单例对象，将 `proximityMonitoringEnabled` 属性设置为 YES 表示设置 UIDevice 开始接收手机距离传感器的信息。设备以距离传感器只有两种状态，一种是近，一种是远。当有物体靠近或远离手机屏幕听筒处到一定距离时，距离传感器会切换状态。不论是距离传感器的状态切换为近距离状态还是切换为远距离状态，系统都会发送 `UIDeviceProximityStateDidChangeNotification` 通知，开发者只需为这个通知添加监听者监听距离传感器状态的变化即可。

实现通知触发的方法 `notice` 如下所示。

```
-(void)notice{
    if ([UIDevice currentDevice].proximityState) {
        NSLog(@"距离近");
    }else{
        NSLog(@"距离远");
    }
}
```

当距离传感器的状态切换为近距离时，手机屏幕会自动暗掉。



提示

关于 UIDevice 类，开发者常用其获取设备的基础信息，其中常用属性示例如下：

```
//获取设备名称 例如“我的 iPhone”
@property(nonatomic,readonly,strong) NSString *name;
//获取设备类型 例如“iPod touch”
@property(nonatomic,readonly,strong) NSString *model;
//获取系统名称 例如“iOS”
@property(nonatomic,readonly,strong) NSString *systemName;
//获取系统版本 例如“9.0”
@property(nonatomic,readonly,strong) NSString *systemVersion;
```

7.4 iOS 蓝牙开发技术

蓝牙是设备近距离通信的一种方便手段，在 iPhone 中的蓝牙协议升级为 4.0 之后，可以和 iPhone 设备进行通信的电子设备更加多样化，开发者也可以更加方便地开发出 iOS 平台上的蓝牙对战游戏。

相比前面的传感器技术，iOS 中蓝牙传感器的开发要复杂的多。与蓝牙相关的接口都由 CoreBluetooth.framework 框架提供。在进行蓝牙通信时，建立连接通信的双方必须以一方作为中心设备，一方作为外围设备。图 7-6 所示为蓝牙技术开发中中心设备与外围设备的关系。

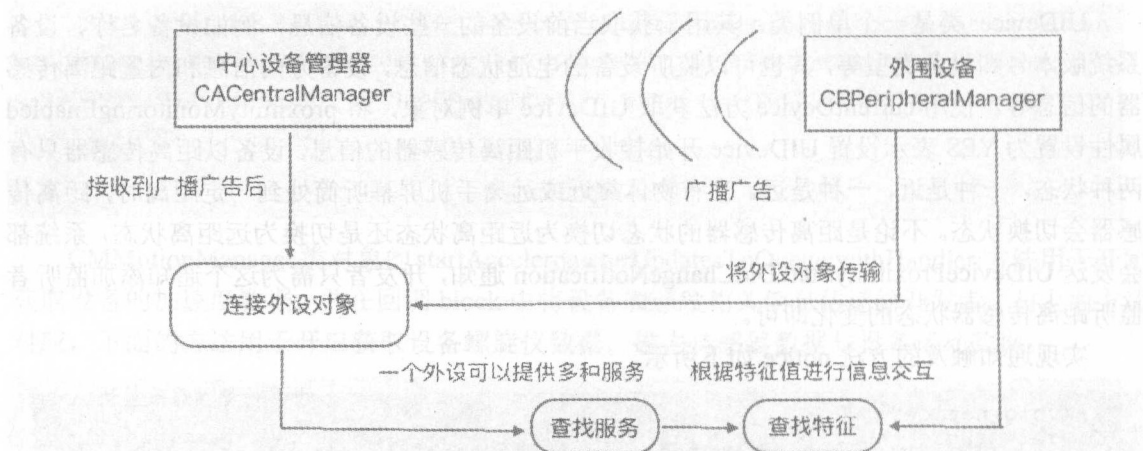


图 7-6 蓝牙开发中中心设备与外围设备的关系

根据图 7-6 进行分析，在建立蓝牙通信连接前，中心设备会作为监听者来扫描周围开启蓝牙的外围设备，外围设备会一直向周围发送广播广告，广告中可以传输外围设备的一些基本信息，中心设备接收到外围设备广播出来的广告数据后，可以根据广告中的信息选择约定的外围设备进行连接，一旦连接建立，中心设备可以从外围设备对象中查询外围设备所提供的服务，同样，外围设备在连接中心设备前，应准备好所能提供的服务。一个外围设备可以提供多种服务，例如一个蓝牙打印机可能会提供打印机信息服务、数据打印服务等。当中心设备查找到其需要的服务后，可以搜索查找服务中提供的特征值，特征值是进行蓝牙数据交换的原单位，设备将要传输交互的数据都封装在特征值中，一个服务中也可以提供多个特征值，例如对中心设备而言，读写服务可以提供两个特征值，一个特征值用于读数据，一个特征值用于写数据。

7.4.1 中心设备管理类 CBCentralManager

由于在蓝牙通信技术中涉及的代理与回调十分多而且繁琐，并且因其只能在真机且需要在两个设备之间进行测试，这对读者的学习带来了一定困难，但是不用灰心，本章节将从基础的中心设备管理类搭建和外围设备管理类搭建开始讲起，使读者可以清楚地了解各个回调函数的调用场景与之间的交互关系。

如上所述，中心设备是建立连接双方中的被动方，其从所有扫描到的外设中找到需要配对的进行连接。使用 Xcode 创建一个名为 CBCentralManagerTest 的工程，在 ViewController.m 文件中导入如下头文件。

```
#import <CoreBluetooth/CoreBluetooth.h>
```

在 ViewController.m 文件中遵守相关协议并且声明一些属性如下所示。

```
@interface ViewController () <CBCentralManagerDelegate, CBPeripheralDelegate>
{
    //中心设备管理对象
    CBCentralManager * _centerManger;
```

```

//要连接的外设
CBPeripheral * _peripheral;
//要交互的外设属性
CBCharacteristic * _chara;
UILabel * _messageLabel;
}
@end

```

上面声明的属性中, `_centerManager` 对象为中心设备管理对象, 用于扫描与协调外设工作。`_peripheral` 对象为进行蓝牙交互的外设对象, `_chara` 为进行交互的特征值, `_messageLabel` 用于在界面显示外设传输过来的数据。

在 `ViewController.m` 的 `viewDidLoad` 方法中添加如下代码。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    _centerManger = [[CBCentralManager alloc] initWithDelegate:self queue:dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)];
    [self creatView];
}

```

`CBCentralManager` 类的 `initWithDelegate:queue:` 方法用于初始化中心设备管理对象, 这个方法中第1个参数为设置实现代理方法的对象, 第2个参数用于设置调用代理方法的线程队列, 这里将其设置为系统提供的一个并行全局队列, 关于线程与 GCD 的相关知识, 后面章节会进行介绍, 这里读者不必过多关注。`creatView` 方法用于初始化 UI 界面。`creatView` 方式实现如下所示。

```

- (void)creatView{
    _messageLabel = [[UILabel alloc] initWithFrame:CGRectMake(20, 100, self.view.frame.size.width-40, 30)];
    _messageLabel.backgroundColor = [UIColor greenColor];
    _messageLabel.textColor = [UIColor redColor];
    _messageLabel.textAlignment = NSTextAlignmentCenter;
    [self.view addSubview:_messageLabel];
}

```

当初始化中心设备管理对象并且设置代理之后, 系统会检测当前设备蓝牙状态的可用性, 检测结果可以在如下代理方法中处理。

```

//设备硬件检测状态回调的方法 可用后开始扫描设备
- (void)centralManagerDidUpdateState:(CBCentralManager *)central{
    if (_centerManger.state==CBCentralManagerStatePoweredOn) {
        [_centerManger scanForPeripheralsWithServices:nil options:nil];
    }
}

```

`centralManagerDidUpdateState`:代理方法在系统检测到当前设备的蓝牙状态改变时会被调用,开发者可以在其中判断用户是否开启了设备的蓝牙功能,CBCentralManager 对象中的 `state` 属性枚举了设备的蓝牙状态,枚举值及意义如下所示。

```
typedef NS_ENUM(NSInteger, CBCentralManagerState) {
    //状态未知
    CBCentralManagerStateUnknown = 0,
    //连接断开将要重置
    CBCentralManagerStateResetting,
    //设备不支持蓝牙
    CBCentralManagerStateUnsupported,
    //为授权使用蓝牙
    CBCentralManagerStateUnauthorized,
    //蓝牙功能关闭状态
    CBCentralManagerStatePoweredOff,
    //蓝牙功能正常开启
    CBCentralManagerStatePoweredOn,
};
```

当检测到设备的蓝牙功能正常开启后,使用 CBCentralManager 类对象调用 `scanForPeripherals-WithServices:options:`方法开启扫描周围外设功能。这个方法中第 1 个参数用于设置扫描提供特定服务的外设,设置为 `nil` 则默认扫描所有外设,第 2 个参数用于设置扫描过程中的一些配置项,如果不需要可以设置为 `nil`,这个字典中支持配置的键值及意义如下所示。

```
//是否允许重复扫描 对应 NSNumber 的 bool 值,默认为 NO,会自动去重
NSString *const CBCentralManagerScanOptionAllowDuplicatesKey;
//要扫描的设备 UUID 数组 对应 NSArray
NSString *const CBCentralManagerScanOptionSolicitedServiceUUIDsKey;
```

开始进行外设扫描后,如果扫描到周围可见的蓝牙设备后,会调用如下代理方法。

```
//发现外设后调用的方法
-(void)centralManager:(CBCentralManager *)central didDiscoverPeripheral:
(CBPeripheral *)peripheral advertisementData:(NSDictionary<NSString *,id> *)ad
vertisementData RSSI:(NSNumber *)RSSI{
    //获取设备的名称 或者广告中的相应字段来配对
    NSString * name = [advertisementData objectForKey:CBAdvertisementDataL
ocalNameKey];
    if ([name isEqualToString:@"ZYH"]){
        //保存此设备
        _peripheral = peripheral;
        //进行连接
    }
}
```



```

        [_centerManger connectPeripheral:peripheral options:@{CBConnectPeripheralOptionNotifyOnConnectionKey:@YES}];
    }
}

```

centralManager:didDiscoverPeripheral:advertisementData:RSSI:方法会在中心设备发现周围外设后调用,这个方法中传进来的第1个参数为发起扫描的中心设备,第2个参数为所发现的外设设备,第3个参数为所发现的外设设备广播的广告信息。开发者在找到所需要配对的外设后,使用中心设备对象调用 **connectPeripheral:options** 方法进行与连接外设。接连外设成功与否的信息会在下面几个代理方法中传递给开发者。

```

//连接断开
-(void)centralManager:(CBCentralManager *)central didDisconnectPeripheral:
(CBPeripheral *)peripheral error:(NSError *)error{
    NSLog(@"连接断开");
    [_centerManger connectPeripheral:peripheral options:@{CBConnectPeripheralOptionNotifyOnConnectionKey:@YES}];
}

//连接设备失败回调的方法
-(void)centralManager:(CBCentralManager *)central didFailToConnectPeripheral:
(CBPeripheral *)peripheral error:(NSError *)error{
    NSLog(@"连接失败");
}

//连接外设成功的回调
-(void)centralManager:(CBCentralManager *)central didConnectPeripheral:(CBPeripheral *)peripheral{
    NSLog(@"连接成功");
    //设置代理与搜索外设中的服务
    [peripheral setDelegate:self];
    [peripheral discoverServices:nil];
}

```

代理方法 **centralManager:didFailToConnectPeripheral:error:**会在连接外设失败后被调用。代理方法 **centralManager:didDisconnectPeripheral:error:**方法会在蓝牙连接断开后被调用,开发者可以在其中进行重连操作。代理方法 **centralManager:didConnectPeripheral:**方法会在连接外设成功后被调用,开发者在这个方法中需要进行外设 **CBPeripheral** 对象代理的设置,并调用 **discoverServices:**方法开始进行外设服务的扫描。

当扫描到外设所提供的服务后,系统会调用如下代理方法。

```

//发现服务后回调的方法
-(void)peripheral:(CBPeripheral *)peripheral didDiscoverServices:(NSError *)error{
    for (CBService *service in peripheral.services)

```

```

{
    //发现服务 比较服务的 UUID
    if ([service.UUID isEqual:[CBUUID UUIDWithString:@"68753A44-4D6F-1
226-9C60-0050E4C00067"]])
    {
        NSLog(@"Service found with UUID: %@", service.UUID);
        //查找服务中的特征值
        [peripheral discoverCharacteristics:nil forService:service];
        break;
    }
}
}

```

开发者可以在 `peripheral:didDiscoverServices:` 方法中遍历所扫描到的外设的所有服务，获取到所需要的服务，调用 `discoverCharacteristics:forService:` 方法进行服务所提供特征值的扫描。当扫描到服务中的特征值后，系统会调用如下代理方法。

```

//开发服务中的特征值后回调的方法
-(void)peripheral:(CBPeripheral *)peripheral didDiscoverCharacteristicsFo
rService:(CBService *)service error:(NSError *)error{
    for (CBCharacteristic *characteristic in service.characteristics)
    {
        //发现特征 比较特征值得 UUID 来获取所需要的
        if ([characteristic.UUID isEqual:[CBUUID UUIDWithString:@"68753A44
-4D6F-1226-9C60-0050E4C00067"]]) {
            //保存特征值
            _chara = characteristic;
            //监听特征值
            [_peripheral setNotifyValue:YES forCharacteristic:_chara];
        }
    }
}

```

开发者可以在 `peripheral:didDiscoverCharacteristicsForService:error:` 方法中遍历服务所提供的所有特征值，找到所需要的特征值进行保存，使用外设对象调用 `setNotifyValue:forCharacteristic:` 方法将对特征值进行订阅，当外设服务中相应特征值发生变化时，系统会回调如下方法通知中心设备。

```

//所监听的特征值更新时回调的方法
- (void)peripheral:(CBPeripheral *)peripheral didUpdateValueForCharacteri
stic:(CBCharacteristic *)characteristic error:(NSError *)error
{

```

```

//更新接收到的数据
NSLog(@"%@", [[NSString alloc] initWithData:characteristic.value encoding:NSUTF8StringEncoding]);
//要在主线程中刷新
dispatch_async(dispatch_get_main_queue(), ^{
    _messageLabel.text = [[NSString alloc] initWithData:characteristic.value encoding:NSUTF8StringEncoding];
});
}

```

代理方法 `peripheral:didUpdateValueForCharacteristic:error:` 在外设服务中的特征值改变时会被调用, 前提是中心设备中连接的外设对象订阅了这个特征值, 在这个方法中, 可以获取到更新后的特征值数据进行逻辑操作, 上面代码将外设更新的特征值通过 `_messageLabel` 显示在屏幕上。

仅仅编写好了中心设备的代码是没办法进行结果测试的, 还需要编写另一个工程用于外设的管理, 外设工程的编写将在一下小节进行介绍。

7.4.2 外围设备管理类 CBPeripheralManager

在上一小节中设计并搭建了一个蓝牙中心设备管理工程, 本小节将以同样的思路设计并搭建外围设备管理类。使用 Xcode 创建一个名为 `CBPeripheralManagerTest` 的工程, 在 `ViewController.m` 文件中导入如下头文件。

```
#import <CoreBluetooth/CoreBluetooth.h>
```

在 `ViewController.m` 文件中遵守相应协议并声明属性如下所示。

```

@interface ViewController () <CBPeripheralManagerDelegate>
{
    //外设管理中心
    CBPeripheralManager * _peripheralManager;
    //外设提供的服务
    CBMutableService * _ser;
    //服务提供的特征值
    CBMutableCharacteristic * _chara;
    //输入文本框
    UITextField * _textField;
    //发送按钮
    UIButton * _sentBtn;
}
@end

```

在 `viewDidLoad` 中进行一些初始化操作如下所示。


```

- (void)viewDidLoad {
    [super viewDidLoad];
    //初始化服务
    _ser= [[CBMutableService alloc] initWithType:[CBUUID UUIDWithString:@"68753A44-4D6F-1226-9C60-0050E4C00067"] primary:YES];
    //初始化特征
    _chara = [[CBMutableCharacteristic alloc] initWithType:[CBUUID UUIDWithString:@"68753A44-4D6F-1226-9C60-0050E4C00067"] properties:CBCharacteristicPropertyNotify value:nil permissions:CBAAttributePermissionsReadable];
    //向服务中添加特征
    _ser.characteristics = @[_chara];
    //初始化外设管理对象
    _peripheralManager = [[CBPeripheralManager alloc] initWithDelegate:self queue:dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)];
    [self creatView];
}

```

在 `viewDidLoad` 方法中进行了外设管理对象、服务及服务特征值的初始化，在 `creatView` 方法中进行了一些界面相关的初始化操作，实现如下所示。

```

- (void)creatView{
    _textField = [[UITextField alloc] initWithFrame:CGRectMake(20, 100, self.view.frame.size.width-40, 30)];
    _textField.borderStyle = UITextBorderStyleRoundedRect;
    [self.view addSubview:_textField];

    _sentBtn = [UIButton buttonWithType:UIButtonTypeSystem];
    _sentBtn.frame= CGRectMake(self.view.frame.size.width/2-30, 150, 60, 30);
    [_sentBtn setTitle:@"发送" forState:UIControlStateNormal];
    [_sentBtn addTarget:self action:@selector(sent) forControlEvents:UIControlEventTouchUpInside];
    _sentBtn.enabled=NO;
    [self.view addSubview:_sentBtn];
}

```

实现按钮的触发方法 `sent` 如下所示。

```

- (void) sent{
    NSData * data = [_textField.text dataUsingEncoding:NSUTF8StringEncoding];
    //将文本框中的文字 更新到特征值中
    [_peripheralManager updateValue:data forCharacteristic:_chara onSubscribedCentrals:nil];
}

```

当外设管理对象初始化完成并且设置了代理后，系统首先会检测设备蓝牙功能的可用性，检测到结果后会回调如下方法。

```
//设备硬件检测状态回到的方法 可用后添加服务与广播广告
-(void)peripheralManagerDidUpdateState:(CBPeripheralManager *)peripheral{
    if (_peripheralManager.state == CBPeripheralManagerStatePoweredOn) {
        //添加服务
        [_peripheralManager addService:_ser];
        //开始广播广告
        [_peripheralManager startAdvertising:@{CBAdvertisementDataLocalNameKey:@"ZYH"}];
    }
}
```

在 `peripheralManagerDidUpdateState:` 中可进行外设管理对象状态的判断，如果判断到设备的蓝牙功能可以正常使用，使用 `addService:` 方法将需要提供的服务添加进外设管理对象，之后调用 `startAdvertising:` 方法进行广播广告。`startAdvertising:` 方法中可以传入一个字典参数，这个字典中可以设置的键值十分严格，只支持对如下的键进行设置。

对应设置 `NSString` 类型的广播名

```
NSString *const CBAdvertisementDataLocalNameKey;
```

外设制造商的 `NSData` 数据

```
NSString *const CBAdvertisementDataManufacturerDataKey;
```

外设制造商的 `CBUUID` 数据

```
NSString *const CBAdvertisementDataServiceDataKey;
```

服务的 `UUID` 与其对应的服务数据字典数组

```
NSString *const CBAdvertisementDataServiceUUIDsKey;
```

附加服务的 `UUID` 数组

```
NSString *const CBAdvertisementDataOverflowServiceUUIDsKey;
```

外设的发送功率 `NSNumber` 类型

```
NSString *const CBAdvertisementDataTxPowerLevelKey;
```

外设是否可以连接

```
NSString *const CBAdvertisementDataIsConnectable;
```

服务的 `UUID` 数组

```
NSString *const CBAdvertisementDataSolicitedServiceUUIDsKey;
```

在上面调用 `addService:` 方法添加服务后，添加服务是否成功的结果会在如下回调方法中传入：

```
//添加服务后回调的方法
-(void)peripheralManager:(CBPeripheralManager *)peripheral didAddService:
(CBService *)service error:(NSError *)error{
    if (error) {
        NSLog(@"添加服务失败");
    }
    NSLog(@"添加服务成功");
}
```

外设进行广播广告后，会首先调用如下代理方法。

```
//开始放广告的回调
-(void)peripheralManagerDidStartAdvertising:(CBPeripheralManager *)periph
heral error:(NSError *)error{
    NSLog(@"播放广告");
}
```

之后，外设会一直等待中心设备的发现与连接，当中心设备发现了外设并且连接了外设订阅了外设提供的服务中的某个特征值后，系统会回调外设管理类的如下代理方法通知外设。

```
//中心设备订阅特征值时回调
-(void)peripheralManager:(CBPeripheralManager *)peripheral central:(CBCent
ral *)central didSubscribeToCharacteristic:(CBCharacteristic *)characteristic{
    //与中心设备订阅成功后 将按钮设置为可用
    _sentBtn.enabled=YES;
    NSLog(@"订阅特征值");
}
```

与上面方法相对，如果中心设备取消了某个特征值的订阅，系统会回调如下方法。

```
//中心设备取消订阅特征值时回调
-(void)peripheralManager:(CBPeripheralManager *)peripheral central:(CBCen
tral *)central didUnsubscribeFromCharacteristic:(CBCharacteristic *)character
istic{
    _sentBtn.enabled=NO;
    NSLog(@"取消订阅特征值");
}
```

通过上面代码的编写，读者已经可以在中心设备与外设间通过蓝牙进行数据交互了，笔者这里使用了 iPhone 与 iPod 进行了测试，其中 iPhone 作为中心设备，iPod 作为外设，在运行上一小节和本小节编写的程序前，要保证设备的蓝牙功能已经开启，两设备之间的交互效果如图 7-7 与图 7-8 所示。



图 7-7 充当蓝牙通信中外设角色的 iPod

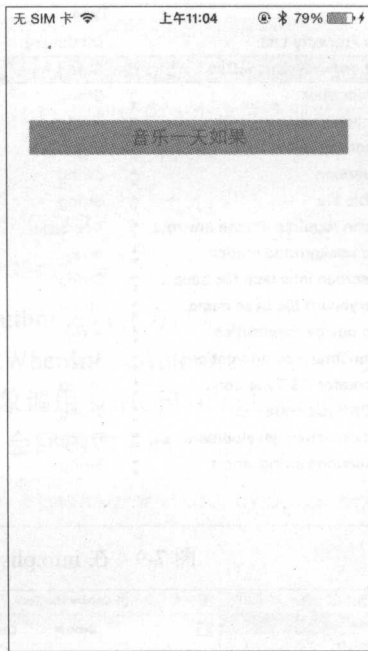


图 7-8 充当蓝牙通信中中心设备角色的 iPhone

7.5 GPS 应用与地图编程技术

iPhone 设备中的 GPS 模块主要用于设备定位和车辆导航等方面,在 iOS 开发中, GPS 模块的功能大多会与地图应用结合使用。在系统原生开发框架中,也包含了地图开发的相关模块,本节将主要介绍 iOS 开发中的设备定位与地图相关的编程技术。

7.5.1 进行设备地理位置定位

定位与地图服务是智能手机方便人们生活的一大特点。在 iOS 设备中,系统对设备定位服务的隐私与权限还是管理的比较严格的,任何正规渠道获取到的应用程序想要获取用户设备的地理位置,都需要经过用户的权限认证。

使用 Xcode 创建一个名为 CoreLocationTest 的工程,若要在工程中使用系统的定位服务,需要在 Info.plist 文件中添加一个键,添加键名为 `NSLocationAlwaysUsageDescription` 则默认无论应用在前台还是后台都会调用定位服务,如果设键名为 `NSLocationWhenInUseUsageDescription` 则只有当应用处于前台时才会调用系统的定位服务。添加此键值后的 Info.plist 文件如图 7-9 所示。

这里需要注意,如果要在前台和后台都调用定位服务,还需要在 Xcode 配置文件的功能页面中做一些配置,配置方式如图 7-10 所示。

Key	Type	Value
▼ Information Property List		
Dictionary (16 items)		
NSLocationAlwaysUsageDes... String		
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
Bundle name	String	\$(PRODUCT_NAME)
InfoDictionary version	String	6.0
Bundle version	String	1
Executable file	String	\$(EXECUTABLE_NAME)
Application requires iPhone enviro...	Boolean	YES
▶ Required background modes	Array	(1 item)
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
▶ Required device capabilities	Array	(1 item)
▶ Supported interface orientations	Array	(3 items)
Bundle creator OS Type code	String	????
Bundle OS Type code	String	APPL
Localization native development re...	String	en
Bundle versions string, short	String	1.0

图 7-9 在 info.plist 文件中添加调用定位服务的键名

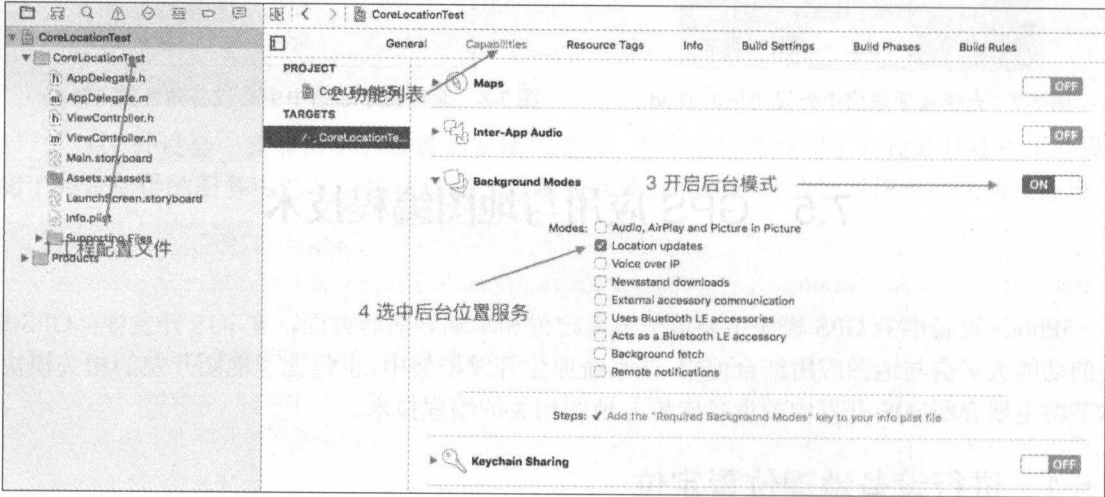


图 7-10 在 Xcode 配置文件中配置后台定位功能

做好了前面的配置工作之后，可以开始进行定位服务代码的编写。首先在 `ViewController.m` 文件中添加如下头文件。

```
#import <CoreLocation/CoreLocation.h>
```

在 `ViewController.m` 中遵守相关协议并声明位置管理对象如下所示。

```
@interface ViewController ()<CLLocationManagerDelegate>
{
    CLLocationManager* manager;
}
@end
```

`CLLocationManager` 类是负责位置管理的类，在 `viewDidLoad` 方法中添加代码如下所示。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    manager = [[CLLocationManager alloc] init]; //初始化一个定位管理对象
    [manager requestAlwaysAuthorization]; //申请定位服务权限
    manager.delegate=self; //设置代理
    [manager startUpdatingLocation]; //开启定位服务
}

```

当 `CLLocationManager` 对象调用 `requestAlwaysAuthorization` 方法后, 会向用户申请前台后台都使用位置服务, 与这个方法相对应, 调用 `requestWhenInUseAuthorization` 方法会向用户申请只在前台使用定位服务。`CLLocationManager` 类对象调用 `startUpdatingLocation` 方法后将开始进行设备定位, 一直到位置或者设备位置改变后, 会回调如下的代理方法:

```

- (void)locationManager:(CLLocationManager *)manager didUpdateLocations:(NSArray<CLLocation *> *)locations{
    NSLog(@"%@", locations);
}

```

`CLLocation` 对象是具体的地理位置信息对象, 其中会封装经纬度、速度和海拔高度等地理信息。



提示

使用模拟器也可以进行模拟定位, 选择模拟器 Debug 菜单中的 Location 选项, 将其设置为 City Bicycle Ride, 模拟器将模拟在城市中进行骑行时的场景, 如图 7-11 所示。

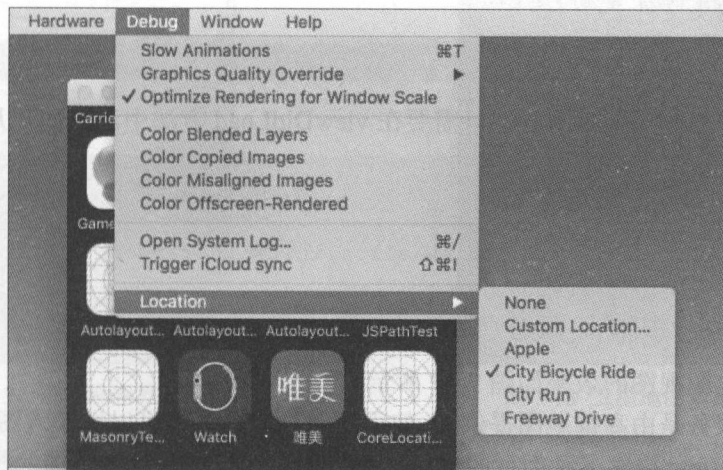


图 7-11 设置模拟器的位置信息

再来看 `CLLocation` 类, 这个类中封装的地理信息常用的有如下几种。

// 设备的经纬度信息

```

@property(readonly, nonatomic) CLLocationCoordinate2D coordinate; //海拔高度, 浮点型

```



```

@property(readonly, nonatomic) CLLocationDistance altitude;
//水平方向的容错半径
@property(readonly, nonatomic) CLLocationAccuracy horizontalAccuracy;
//竖直方向的容错半径
@property(readonly, nonatomic) CLLocationAccuracy verticalAccuracy;
//设备前进的方向, 取值范围为 0~359.9, 相对正北方向
@property(readonly, nonatomic) CLLocationDirection course;
//速度, 单位为 m/s
@property(readonly, nonatomic) CLLocationSpeed speed;
//定位时的时间戳
@property(readonly, nonatomic, copy) NSDate *timestamp;

```

关于经纬度信息属性 `coordinate`, 其是一个结构体, 如下所示。

```

typedef struct {
    CLLocationDegrees latitude;//纬度
    CLLocationDegrees longitude;//经度
} CLLocationCoordinate2D;

```

7.5.2 原生地图开发技术

iOS 开发中, 定位技术往往会和地图编程技术结合使用, 本小节将向读者介绍向应用中添加系统地图服务的相关方法。使用 Xcode 创建一个名为 `MapKitTest` 的工程, iOS 系统中地图开发的相关接口与方法都封装在 `MapKit.framework` 这个框架中, 在工程的 `ViewController.m` 文件中导入如下的头文件。

```
#import <MapKit/MapKit.h>
```

接入地图界面的方法十分简单, 只需要在 `viewDidLoad` 方法中添加如下几行代码。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    MKMapView * map = [[MKMapView alloc] initWithFrame:self.view.frame];
    [self.view addSubview:map];
}

```

运行工程, 地图视图的效果如图 7-12 所示。

系统的地图服务是由高德地图提供的, 使用捏合与双击手势可以操作地图进行缩放和改变比例尺。MKMapView 提供多种类型的地图, 可以通过如下方法设置地图的模式。

```
map.mapType = MKMapTypeHybridFlyover;
```

`mapType` 属性需要设置为 `MKMapType` 类型的枚举值, 可选的枚举值及意义如下所示。

```

typedef NS_ENUM(NSUInteger, MKMapType) {
    //标准行政地图
    MKMapTypeStandard = 0,

```

```
//标准卫星地图
MKMapTypeSatellite,
//行政与卫星混合地图
MKMapTypeHybrid,
//立体卫星地图 iOS 9 后可用
MKMapTypeSatelliteFlyover NS_ENUM_AVAILABLE(10_11, 9_0),
//立体混合地图 iOS 9 后可用
MKMapTypeHybridFlyover NS_ENUM_AVAILABLE(10_11, 9_0),
} NS_ENUM_AVAILABLE(10_9, 3_0) __WATCHOS_PROHIBITED;
```

有关卫星地图的效果如图 7-13 所示。

可以使用如下代码方法设置地图的中心位置和比例尺。

```
map.region= MKCoordinateRegionMake(CLLocationCoordinate2DMake(39.26, 116.3), MKCoordinateSpanMake(1.8, 2.05));
```

MKCoordinateRegionMake()方法有两个参数,第1个参数设置经纬度坐标,第2个参数设置地图界面显示的经纬度范围,运行工程,效果如图 7-14 所示。



图 7-12 系统地图界面

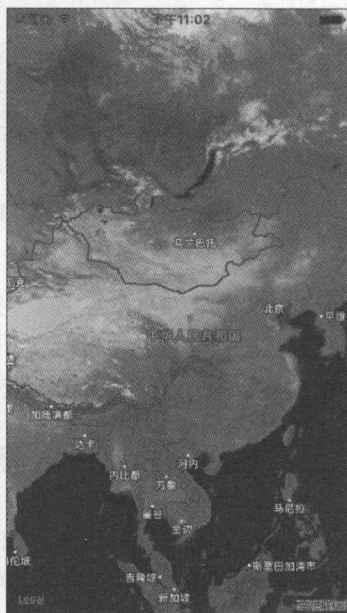


图 7-13 卫星地图效果图

MapKit 地图开发框架中也集成了定位用户位置的服务,根据前面小节在工程中配置好 info.plist 文件并使用 CLLocationManager 申请到位置服务权限后,可以调用如下的方法来在地图上地位用户的位置。

```
//是否显示用户位置
map.showsUserLocation = YES;
//用户位置的追踪模式
map.userTrackingMode = MKUserTrackingModeFollow;
```

上面代码中，设置 `showsUserLocation` 属性为 YES 将开启在地图上显示用户位置的功能，设置 `userTrackingMode` 属性将决定用户位置追踪的模式，可设置的枚举值及意义如下所示。

```
typedef NS_ENUM(NSInteger, MKUserTrackingMode) {
    MKUserTrackingModeNone = 0, // 不追踪用户位置
    MKUserTrackingModeFollow, // 追踪用户位置
    MKUserTrackingModeFollowWithHeading, // 用户方向改变时追踪位置
};
```

运行工程，在地图上可以看到一个蓝点标出了用户所在的地理位置，如图 7-15 所示。

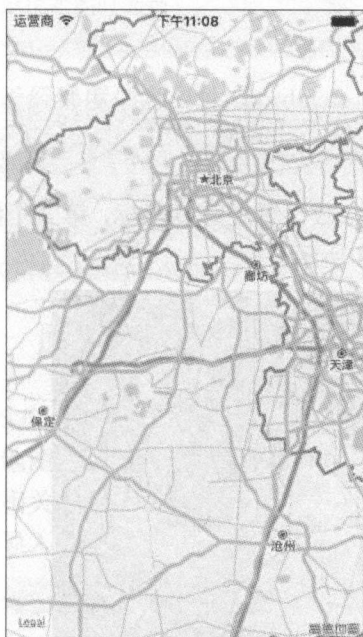


图 7-14 设置地图中心位置与显示比例尺

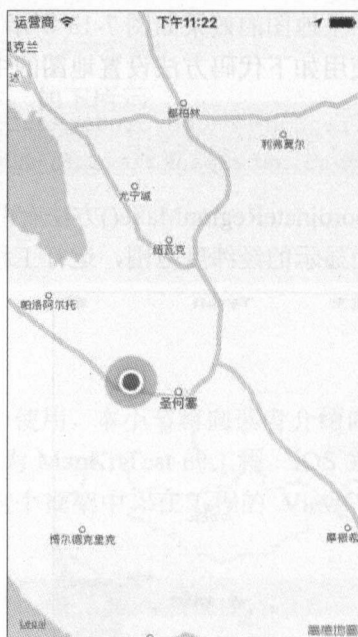


图 7-15 在地图上追踪用户位置

7.5.3 在地图中添加大头针及标注

地图中往往需要添加一些标注，例如在使用路线导航时，地图上会被标注许多大头针标签，每个标签都是路线中的转折点。本小节将介绍在地图中添加大头针标注的相关方法。

打开创建的 MapKitTest 工程，在 `ViewController.m` 文件的 `viewDidLoad` 中添加如下代码。

```
MKPointAnnotation * ann = [[MKPointAnnotation alloc] init];
ann.coordinate = CLLocationCoordinate2DMake(39.26, 116.3);
ann.title = @"我的位置";
ann.subtitle = @"中华人民共和国北京";
[map addAnnotation:ann];
```

`MKPointAnnotation` 类用于创建地图上的大头针标注，`coordinate` 属性设置标注位置的经纬度，`title` 属性设置单击标注后显示的标题，`subtitle` 属性设置单击标注后显示 IDE 副标题，

MKMapView 类对象的 addAnnotation:方法用于向地图视图中添加一个大头针标注,运行工程,效果如图 7-16 所示。

当向 MKMapView 视图中添加大头针控件时,是会回调一个代理方法的,开发者可以在这个方法中进行大头针颜色的自定义,首先在 ViewController.m 文件中遵守如下协议。

```
@interface ViewController ()<MKMapViewDelegate>
{
    CLLocationManager *manager;
}
@end
```

在 viewDidLoad 方法中设置 MKMapView 视图的代理。

```
map.delegate=self;
```

在 ViewController.m 文件中实现如下方法。

```
-(MKAnnotationView *)mapView:(MKMapView *)mapView viewForAnnotation:(id<MKAnnotation>)annotation{
    MKPinAnnotationView * view = [[MKPinAnnotationView alloc] initWithAnnotation:annotation reuseIdentifier:@"id"];
    view.pinTintColor = [UIColor purpleColor];
    return view;
}
```

MKPinAnnotation 类用于自定义大头针的样式等 UI 效果,上面代码将大头针的颜色设置成了紫色,效果如图 7-17 所示。



图 7-16 在地图上添加大头针标注



图 7-17 自定义大头针标注的颜色

当单击大头针时，大头针上会弹出信息视图，其实开发者也可以对这个信息视图及大头针的图片进行 UI 上的自定义，在 `mapView:viewForAnnotation:` 方法编写如下代码。

```
-(MKAnnotationView *)mapView:(MKMapView *)mapView viewForAnnotation:(id<MKAnnotation>)annotation{
    MKAnnotationView * view = [[MKAnnotationView alloc] initWithAnnotation:
    annotation reuseIdentifier:@"id"];
    view.image = [UIImage imageNamed:@"image"];
    //设置开启详情视图
    view.canShowCallout=YES;
    UIView * view1 = [[UIView alloc] initWithFrame:CGRectMake(0, 0, 50, 50)];
    view1.backgroundColor=[UIColor redColor];
    UIView * view2 = [[UIView alloc] initWithFrame:CGRectMake(0, 0, 30, 50)];
    view2.backgroundColor=[UIColor blueColor];
    //设置左右辅助视图
    view.leftCalloutAccessoryView=view1;
    view.rightCalloutAccessoryView=view2;
    //设置拖拽，可以通过单击不放进行拖拽
    view.draggable=YES;
    return view;
}
```

需要注意，如果设置 `MKAnnotationView` 类对象的 `draggable` 属性为 YES，则用户可以通过拖动的方式改变标注在地图视图上的位置，上面的代码运行效果如图 7-18 所示。



图 7-18 完全自定义的地图标注视图

7.5.4 在地图视图中添加覆盖物

地图覆盖物就是在地图上添加一些类似路径、区域、图形等物件，其添加与使用原理与大头针的应用十分相似。

系统提供的地图覆盖物主要有 3 种，分别为折线、圆和多边形。为地图添加折线覆盖物在 ViewContrller.m 文件的 viewDidLoad 方法中添加如下代码。

```
//下面是 c 的语法，创建一个结构体数组
CLLocationCoordinate2D *coor;
coor = malloc(sizeof(CLLocationCoordinate2D)*5);
for (int i=0; i<5; i++) {
    CLLocationCoordinate2D po = CLLocationCoordinate2DMake(33.23+i*0.01,
113.112+(i%2)*0.01);
    coor[i]=po;
}
//创建一个折线对象
MKPolyline * line = [MKPolyline polylineWithCoordinates:coor count:5];
//释放指针
free(coor);
coor=nil;
[map addOverlay:line];
```

上面代码创建了一组折线转折点坐标，在 ViewController.m 文件中实现如下代理方法对折线覆盖物的属性进行设置。

```
//覆盖物绘制的代理
-(MKOverlayRenderer *)mapView:(MKMapView *)mapView rendererForOverlay:(id
<MKOverlay>)overlay{
    //折线覆盖物提供类
    MKPolylineRenderer * render = [[MKPolylineRenderer alloc] initWithPolyline:overlay];
    //设置线宽
    render.lineWidth=3;
    //设置颜色
    render.strokeColor=[UIColor redColor];
    return render;
}
```

运行工程，效果如图 7-19 所示。

在 viewDidLoad 方法中使用如下代码可以添加圆形覆盖物。

```
MKCircle * circle = [MKCircle circleWithCenterCoordinate:CLLocationCoordinate2DMake(34.23, 113.122) radius:500];
[map addOverlay:circle];
```


在 `mapView:rendererForOverlay:` 代理方法中编写如下代码进行圆形覆盖物的设置。

```
//覆盖物绘制的代理
-(MKOverlayRenderer *)mapView:(MKMapView *)mapView rendererForOverlay:(id
<MKOverlay>)overlay{
    MKCircleRenderer * render=[[MKCircleRenderer alloc] initWithCircle:ove
rlay];
    render.lineWidth=3;
    //填充颜色
    render.fillColor=[UIColor greenColor];
    //线条颜色
    render.strokeColor=[UIColor redColor];
    return render;
}
```

运行工程，效果如图 7-20 所示。



图 7-19 在地图视图中添加折线覆盖物

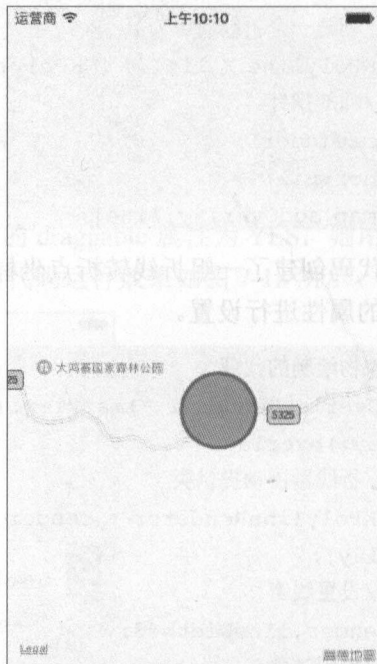


图 7-20 向地图视图中添加圆形覆盖物

在 `viewDidLoad` 方法中添加如下方法来为地图视图添加多边形覆盖物。

```
CLLocationCoordinate2D *coor;
coor = malloc(sizeof(CLLocationCoordinate2D)*6);
for (int i=0; i<5; i++) {
    CLLocationCoordinate2D po = CLLocationCoordinate2DMake(33.23+i*0.01,
113.112+((i/2==0)?0.01:-0.01));
    coor[i]=po;
}
```

```

}
coord[5]=CLLocationCoordinate2DMake(33.23, 113.112);
MKPolygon * gon = [MKPolygon polygonWithCoordinates:coord count:6];
free(coord);
coord=nil;
[map addOverlay:gon];

```

实现相关代理方法如下所示。

```

//覆盖物绘制的代理
-(MKOverlayRenderer *)mapView:(MKMapView *)mapView rendererForOverlay:(id
<MKOverlay>)overlay{
    MKPolygonRenderer * render = [[MKPolygonRenderer alloc] initWithPolygon:
overlay];
    render.lineWidth=3;
    render.strokeColor=[UIColor redColor];
    return render;
    return render;
}

```

运行工程，效果如图 7-21 所示。

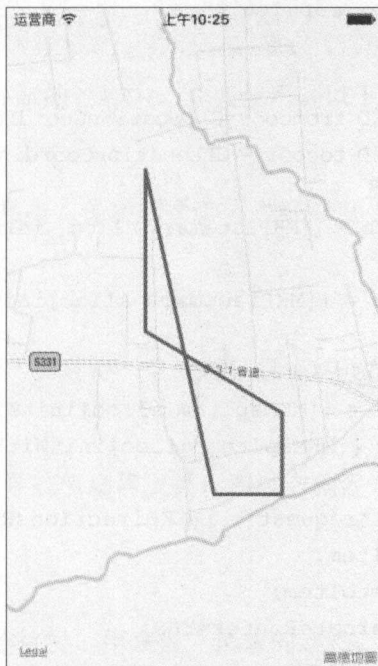


图 7-21 在地图视图上添加多边形覆盖物

7.5.5 在地图中进行线路导航与附近兴趣点检索

MapKit 框架中也封装了线路导航与附近兴趣点检索的相关功能，这些服务的数据也是由

高德地图提供，使用 Xcode 创建一个名为 MapKitTestTwo 的工程，在 ViewController.m 文件中导入 MapKit 框架的头文件如下所示。

```
#import <MapKit/MapKit.h>
```

在 ViewController.m 文件中遵守相关协议并声明属性如下所示。

```
@interface ViewController ()<MKMapViewDelegate>
{
    MKMapView * mapView;
}
@end
```

在 viewDidLoad 方法中编写如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    mapView = [[MKMapView alloc] initWithFrame:self.view.frame];
    mapView.region=MKCoordinateRegionMake (CLLocationCoordinate2DMake (39.26,
116.3), MKCoordinateSpanMake (5, 5));
    mapView.mapType=MKMapTypeStandard;
    mapView.delegate=self;
    [self.view addSubview:mapView];

    //导航设置
    CLLocationCoordinate2D fromcoor=CLLocationCoordinate2DMake (39.26, 116.3);
    CLLocationCoordinate2D tocoor = CLLocationCoordinate2DMake (33.33, 113.33);
    //创建出发点和目的点信息
    MKPlacemark *fromPlace = [[MKPlacemark alloc] initWithCoordinate:fromcoor
addressDictionary:nil];
    MKPlacemark *toPlace = [[MKPlacemark alloc] initWithCoordinate:tocoor a
ddressDictionary:nil];
    //创建出发节点和目的地节点
    MKMapItem * fromItem = [[MKMapItem alloc] initWithPlacemark:fromPlace];
    MKMapItem * toItem = [[MKMapItem alloc] initWithPlacemark:toPlace];
    //初始化导航搜索请求
    MKDirectionsRequest *request = [[MKDirectionsRequest alloc] init];
    request.source=fromItem;
    request.destination=toItem;
    request.requestsAlternateRoutes=YES;
    //初始化请求检索
    MKDirections *directions = [[MKDirections alloc] initWithRequest:request];
    //开始检索，结果会返回在 block 中
    [directions calculateDirectionsWithCompletionHandler:^(MKDirectionsRe
sponse *response, NSError *error) {
        if (error) {
```



```
@interface MKDirectionsResponse : NSObject
@property (nonatomic, readonly) MKMapItem *source; //起点
@property (nonatomic, readonly) MKMapItem *destination; //终点
@property (nonatomic, readonly) NSArray *routes; //线路规划数组
@end
```

6. MKETResponse

MKETResponse 封装了线路导航中与时间相关的信息。

7. MKRoute

MKRoute 是线路类，进行起点到目的地的线路导航时，服务有可能会提供多条导航线路，每条导航线路都是一个 MKRoute 对象，这个类中属性如下所示。

```
@interface MKRoute : NSObject
@property (nonatomic, readonly) NSString *name; //线路名称
@property (nonatomic, readonly) NSArray *advisoryNotices; //注意事项
@property (nonatomic, readonly) CLLocationDistance distance; //距离
@property (nonatomic, readonly) NSTimeInterval expectedTravelTime; //耗时
@property (nonatomic, readonly) MKDirectionsTransportType transportType;
//检索的类型
@property (nonatomic, readonly) MKPolyline *polyline; // 线路覆盖物
@property (nonatomic, readonly) NSArray *steps; // 线路详情数组
@end
```

8. MKRouteStep

MKRouteStep 类为导航线路中具体每一节点信息的包装类，其中属性如下所示。

```
@interface MKRouteStep : NSObject
@property (nonatomic, readonly) NSString *instructions; // 节点信息
@property (nonatomic, readonly) NSString *notice; // 注意事项
@property (nonatomic, readonly) MKPolyline *polyline; //线路覆盖物
@property (nonatomic, readonly) CLLocationDistance distance; // 距离
@property (nonatomic, readonly) MKDirectionsTransportType transportType;
// 导航类型
@end
```

在 ViewController.m 文件中实现定义大头针视图与覆盖物视图的方法如下所示。

```
//地图覆盖物的代理方法
-(MKOverlayRenderer *)mapView:(MKMapView *)mapView rendererForOverlay:(id
<MKOverlay>)overlay{
    MKPolylineRenderer *renderer = [[MKPolylineRenderer alloc] initWithPolyline:overlay];
    renderer.strokeColor = [UIColor redColor];
    renderer.lineWidth = 4.0;
```

```

    return renderer;
}
//标注的代理方法
-(MKAnnotationView *)mapView:(MKMapView *)mapView viewForAnnotation:(id<MKAnnotation>)annotation{
    MKPinAnnotationView * view= [[MKPinAnnotationView alloc] initWithAnnotation:annotation reuseIdentifier:@"anno"];
    view.canShowCallout=YES;
    return view;
}

```

运行工程，效果如图 7-22 所示，单击地图中的大头针标注，会显示节点的相关导航信息。

所有的地图类应用都会有附近兴趣点检索这样的功能，例如，搜索附近所有的咖啡店和搜索附近所有的旅社等，iOS 原生的 MapKit 框架也支持进行附近兴趣点检索的操作，使用如下代码。

```

CLLocationCoordinate2D tocoor = CLLocationCoordinate2DMake(33.33, 113.33);
//创建一个位置信息对象，第一个参数为经纬度，第二个为纬度检索范围，单位为米，第三个为经度检索范围，单位为米
MKCoordinateRegion region = MKCoordinateRegionMakeWithDistance(tocoor,
5000, 5000);
//初始化一个检索请求对象
MKLocalSearchRequest * req = [[MKLocalSearchRequest alloc] init];
//设置检索参数
req.region=region;
//兴趣点关键字
req.naturalLanguageQuery=@"school";
//初始化检索
MKLocalSearch * ser = [[MKLocalSearch alloc] initWithRequest:req];
//开始检索，结果返回在 block 中
[ser startWithCompletionHandler:^(MKLocalSearchResponse *response, NSError *error) {
    //兴趣点节点数组
    NSArray * array = [NSArray arrayWithArray:response.mapItems];
    for (int i=0; i<array.count; i++) {
        MKMapItem * item=array[i];
        MKPointAnnotation * point = [[MKPointAnnotation alloc] init];
        point.title=item.name;
        point.subtitle=item.phoneNumber;
        point.coordinate=item.placemark.coordinate;
        [mapView addAnnotation:point];
    }
}];

```


上面代码中，MKLocalSearch 类用于创建一个附近兴趣点的检索对象，regionsh 属性设置要检索的区域范围，naturalLanguageQuery 设置检索兴趣点的关键字。运行工程，效果如图 7-23 所示。

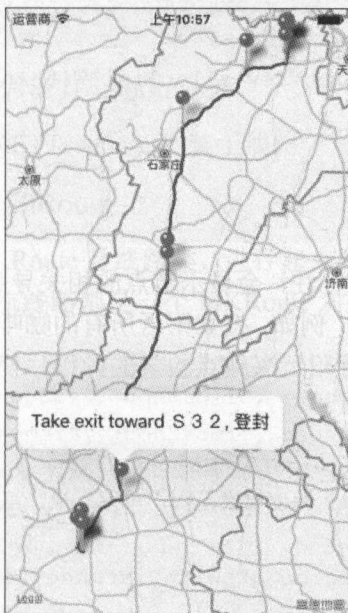


图 7-22 在地图中添加线路导航



图 7-23 在地图上检索周围兴趣点

7.6 实战：简易蓝牙对战五子棋

在前边的小节中，向读者介绍了 iOS 系统蓝牙通信开发的基础知识，前边介绍的蓝牙开发相关知识更多偏向于理论的讲解，所做的演示也是通过两个设备（一个作为中心设备，一个作为外设）来独立进行的。然而在实际应用中，尤其是蓝牙对战的游戏软件中，用户往往既可以作为中心设备，也可以作为外设，本节实战将综合运用蓝牙技术来开发一款简易的对战五子棋小游戏，使读者能够在开发中更加熟练地使用技术。

7.6.1 游戏核心通信类的设计

作为蓝牙对战游戏，其核心就在于设备间的连接与通信，使用 Xcode 创建一个名为 BlueGame 的工程，在其中创建一个继承于 NSObject 的类，命名为 BlueToothTool，将它作为游戏通信和核心工具类。首先在 BlueToothTool.h 文件中添加如下头文件。

```
#import <UIKit/UIKit.h>
#import <CoreBluetooth/CoreBluetooth.h>
```

在 BlueToothTool.h 文件中创建一个协议，用于处理接收到所连接设备发送来的信息之后的回调。

```
@protocol BluetoothToolDelegate <NSObject>
//获取对方数据
-(void)getData:(NSString *)data;
@end
```

在 BluetoothTool.h 中声明如下属性与方法。

```
@interface BluetoothTool : NSObject<CBPeripheralManagerDelegate,CBCentral
ManagerDelegate,CBPeripheralDelegate,UIAlertViewDelegate>
//代理
@property(nonatomic,weak)id<BluetoothToolDelegate>delegate;
//标记是否是房主
@property(nonatomic,assign)BOOL isCentral;
/**
 *获取单例对象的方法
 */
+(instancetype)sharedManager;
/*
 *作为游戏的房主建立游戏房间
 */
-(void)setUpGame:(NSString *)name block:(void(^)(BOOL first))finish;
/*
 *作为游戏的加入者查找附近的游戏
 */
-(void)searchGame;
/**
 *断块连接
 */
-(void)disConnect;
/*
 *进行写数据操作
 */
-(void)writeData:(NSString *)data;
@end
```

在 BluetoothTool.m 文件中声明如下私有属性。

```
@implementation BluetoothTool
{
    //外设管理中心
    CBPeripheralManager * _peripheralManager;
    //外设提供的服务
    CBMutableService * _ser;
    //服务提供的读特征值
    CBMutableCharacteristic * _readChara;
```

```

//服务提供的写特征值
CBMutableCharacteristic * _writeChara;
//等待对方加入的提示视图
UIView * _waitOtherView;
//正在扫描附近游戏的提示视图
UIView * _searchGameView;
//设备中心管理对象
CBCentralManager * _centerManger;
//要连接的外设
CBPeripheral * _peripheral;
//要交互的外设属性
CBCharacteristic * _centerReadChara;
CBCharacteristic * _centerWriteChara;
//处理开始游戏的回调 block
void(^block) (BOOL first);
}

```

实现 `BluetoothTool` 类的单例方法如下所示。

```

//实现单例方法
+(instancetype)sharedManager{
    static BluetoothTool *tool = nil;
    static dispatch_once_t predicate;
    dispatch_once(&predicate, ^{
        tool = [[self alloc] init];
    });
    return tool;
}

```

先来实现作为游戏房主创建游戏的相关方法，将游戏中的房主与客人映射到蓝牙通信中实际的逻辑是这样的：游戏中的房主创建房间后，相当于蓝牙通信中的外设，它将广播自己的存在，等待其他设备的加入；游戏中的客人在蓝牙通信中实际充当中心设备的角色，搜索到外设的广播后与外设进行连接及加入游戏操作。实现创建游戏的方法 `setUpGame:block:` 如下所示。

```

//实现创建游戏的方法
-(void)setUpGame:(NSString *)name block:(void (^)(BOOL))finish{
    block = [finish copy];
    if (_peripheralManager==nil) {
        //初始化服务
        _ser= [[CBMutableService alloc] initWithType:[CBUUID UUIDWithString:@"68753A44-4D6F-1226-9C60-0050E4C00067"] primary:YES];
        //初始化特征
        _readChara = [[CBMutableCharacteristic alloc] initWithType:[CBUUID UUIDWithString:@"68753A44-4D6F-1226-9C60-0050E4C00067"] properties:CBCharacteristicPropertyNotify value:nil permissions:CBAAttributePermissionsReadable];
    }
}

```



```

        _writeChara = [[CBMutableCharacteristic alloc] initWithType:[CBUUID
        UUIDWithString:@"68753A44-4D6F-1226-9C60-0050E4C00068"] properties:CBCharac
        teristicPropertyWriteWithoutResponse value:nil permissions:CBAAttributePermis
        sionsWriteable];
        //向服务中添加特征
        _ser.characteristics = @[_readChara, _writeChara];
        _peripheralManager = [[CBPeripheralManager alloc] initWithDelegate:
        self queue:dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)];
    }
    //设置为房主
    _isCentral=YES;
    //开始广播广告
    [_peripheralManager startAdvertising:@{CBAdvertisementDataLocalNameKe
    y:@"WUZIgame"}];
}

```

实现检测蓝牙可用状态的回调方法如下所示。

```

//外设检测蓝牙状态
-(void)peripheralManagerDidUpdateState:(CBPeripheralManager *)peripheral{
    //判断是否可用
    if (peripheral.state==CBPeripheralManagerStatePoweredOn) {
        //添加服务
        [_peripheralManager addService:_ser];
        //开始广播广告
        [_peripheralManager startAdvertising:@{CBAdvertisementDataLocalNam
        eKey:@"WUZIgame"}];
    }else{
        //弹提示框
        dispatch_async(dispatch_get_main_queue(), ^{
            [self showAlert];
        });
    }
}

```

在上面的方法中，如果检测到蓝牙设备可用则开始广播广告，如果发现蓝牙设备不可用，则弹出提示框提示用户，showAlert方法实现如下所示，这里采用了兼容性比较好的 UIAlertView 来创建。

```

//弹提示框的方法
-(void)showAlert{
    UIAlertView * alert = [[UIAlertView alloc] initWithTitle:@"温馨提示" mes
    sage:@"请确保您的蓝牙可用" delegate:nil cancelButtonTitle:@"好的" otherButtonTit
    les:nil, nil];
}

```

```
[alert show];
}
```

在外设开始广播广告的回调代理方法中编写如下代码。

```
//开始放广告的回调
-(void)peripheralManagerDidStartAdvertising:(CBPeripheralManager *)peripheral error:(NSError *)error{
    if (_waitOtherView==nil) {
        _waitOtherView = [[UIView alloc] initWithFrame:CGRectMake([UIScreen mainScreen].bounds.size.width/2-100, 240, 200, 100)];
        dispatch_async(dispatch_get_main_queue(), ^{
            UILabel * label = [[UILabel alloc] initWithFrame:CGRectMake(0, 0, 200, 100)];
            label.backgroundColor = [UIColor clearColor];
            label.textAlignment = NSTextAlignmentCenter;
            label.text = @"等待附近玩家加入";
            [_waitOtherView addSubview:label];
            _waitOtherView.backgroundColor = [UIColor colorWithRed:0 green:0 blue:0 alpha:0.4];
            [[[UIApplication sharedApplication].delegate window]addSubview:_waitOtherView];
        });
    }else{
        dispatch_async(dispatch_get_main_queue(), ^{
            [_waitOtherView removeFromSuperview];
            [[[UIApplication sharedApplication].delegate window]addSubview:_waitOtherView];
        });
    }
}
```

调用了上面广播广告的方法后，程序的主动权实际上就交由了对方设备，直到和对方设备建立了连接，程序逻辑才会继续向下执行，实现特征值被监听后的回调方法如下所示。

```
//中心设备订阅特征值时回调
-(void)peripheralManager:(CBPeripheralManager *)peripheral central:(CBCentral *)central didSubscribeToCharacteristic:(CBCharacteristic *)characteristic{
    [_peripheralManager stopAdvertising];
    if (_isCentral) {
        UIAlertView * alert = [[UIAlertView alloc] initWithTitle:@" " message:@"请选择先手后手" delegate:self cancelButtonTitle:@"我先手" otherButtonTitles:@"我后手", nil];
        dispatch_async(dispatch_get_main_queue(), ^{
            [_waitOtherView removeFromSuperview];
        });
    }
}
```



```

        [alert show];
    });
}
}

```

上面方法的意义为当对方设备监听了特征值后,证明对方设备已经准备就绪,此时程序中将弹出先手后手的选择提供给用户,实现警告框按钮单击后的回调方法如下所示。

```

-(void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)
buttonIndex{
    //告诉开发者先后手信息
    if (buttonIndex==0) {
        if (_isCentral) {
            block(1);
        }else{
            block(0);
        }
    }else{
        if (_isCentral) {
            block(0);
        }else{
            block(1);
        }
    }
}
}

```

实现蓝牙外设收到数据的回调方法如下所示。

```

//收到写消息后的回调
-(void)peripheralManager:(CBPeripheralManager *)peripheral didReceiveWriteRequests:(NSArray<CBATTRequest *> *)requests{
    dispatch_async(dispatch_get_main_queue(), ^{
        [self.delegate getData:[NSString alloc] initWithData:requests.firstObject.value encoding:NSUTF8StringEncoding]];
    });
}

```

上面所实现的方法都是针对作为房主创建游戏一方的逻辑,下面将实现作为客人加入游戏的一方的相关逻辑,首先实现搜索附近游戏的方法如下所示。

```

-(void)searchGame{
    if (_centerManger==nil) {
        _centerManger = [[CBCentralManager alloc] initWithDelegate:self queue:dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)];
    }else{
        [_centerManger scanForPeripheralsWithServices:nil options:nil];
    }
}

```



```

        if (_searchGameView==nil) {
            _searchGameView = [[UIView alloc] initWithFrame:CGRectMake([UIScreen mainScreen].bounds.size.width/2-100, 240, 200, 100)];
            UILabel * label = [[UILabel alloc] initWithFrame:CGRectMake(0, 0, 200, 100)];
            label.backgroundColor = [UIColor clearColor];
            label.textAlignment = NSTextAlignmentCenter;
            label.text = @"正在扫描加入描附近游戏";
            _searchGameView.backgroundColor = [UIColor colorWithRed:0 green:0 blue:0 alpha:0.4];
            [_searchGameView addSubview:label];
            [[[UIApplication sharedApplication].delegate window]addSubview:_searchGameView];
        }else{
            [_searchGameView removeFromSuperview];
            [[[UIApplication sharedApplication].delegate window]addSubview:_searchGameView];
        }
    }
    //设置为游戏加入方
    _isCentral = NO;
}

```

实现蓝牙通信中心设备检测蓝牙状态的回调如下所示。

```

//设备硬件检测状态回调的方法 可用后开始扫描设备
-(void)centralManagerDidUpdateState:(CBCentralManager *)central{
    if (_centerManger.state==CBCentralManagerStatePoweredOn) {
        [_centerManger scanForPeripheralsWithServices:nil options:nil];
        if (_searchGameView==nil) {
            dispatch_async(dispatch_get_main_queue(), ^{
                _searchGameView = [[UIView alloc] initWithFrame:CGRectMake([UIScreen mainScreen].bounds.size.width/2-100, 240, 200, 100)];
                UILabel * label = [[UILabel alloc] initWithFrame:CGRectMake(0, 0, 200, 100)];
                label.backgroundColor = [UIColor clearColor];
                label.textAlignment = NSTextAlignmentCenter;
                label.text = @"正在扫描加入描附近游戏";
                _searchGameView.backgroundColor = [UIColor colorWithRed:0 green:0 blue:0 alpha:0.4];
                [_searchGameView addSubview:label];
                [[[UIApplication sharedApplication].delegate window]addSubview:_searchGameView];
            });
        }
    }
}

```

```

    }else{

        dispatch_async(dispatch_get_main_queue(), ^{
            [_searchGameView removeFromSuperview];
            [[[UIApplication sharedApplication].delegate window]addSubview:
            _searchGameView];
        });
    }
}
}else{
    dispatch_async(dispatch_get_main_queue(), ^{
        [self showAlert];
    });
}
}
}

```

实现发现外设后的回调方法如下所示。

```

//发现外设后调用的方法
-(void)centralManager:(CBCentralManager *)central didDiscoverPeripheral:
(CBPeripheral *)peripheral advertisementData:(NSDictionary<NSString *,id> *)ad
vertisementData RSSI:(NSNumber *)RSSI{
    //获取设备的名称 或者广告中的相应字段来配对
    NSString * name = [advertisementData objectForKey:CBAdvertisementDataL
ocalNameKey];
    if ([name isEqualToString:@"WUZIGame"]){
        //保存此设备
        _peripheral = peripheral;
        //进行连接
        [_centerManger connectPeripheral:peripheral options:@{CBConnectPer
ipheralOptionNotifyOnConnectionKey:@YES}];
    }
}
}

```

实现连接设备成功后的回调方法如下所示，在其中进行服务的搜索。

```

//连接外设成功的回调
-(void)centralManager:(CBCentralManager *)central didConnectPeripheral:(C
BPeripheral *)peripheral{
    NSLog(@"连接成功");
    //设置代理与搜索外设中的服务
    [peripheral setDelegate:self];
    [peripheral discoverServices:nil];
    dispatch_async(dispatch_get_main_queue(), ^{
        [_searchGameView removeFromSuperview];
    });
}
}

```


如果因信号异常造成连接中断，程序应该具备自动重连的功能，在连接断开的回调方法中编写如下代码。

```
//连接断开
-(void)centralManager:(CBCentralManager *)central didDisconnectPeripheral:
(CBPeripheral *)peripheral error:(NSError *)error{
    NSLog(@"连接断开");
    [_centerManger connectPeripheral:peripheral options:@{CBCentralManagerOptionNotifyOnConnectionKey:@YES}];
}
```

实现发现服务的回调方法如下所示，注意这里的服务 UUID 要和外设所约定的一致。

```
//发现服务后回调的方法
-(void)peripheral:(CBPeripheral *)peripheral didDiscoverServices:(NSError
*)error{
    for (CBService *service in peripheral.services)
    {
        //发现服务 比较服务的 UUID
        if ([service.UUID isEqual:[CBUUID UUIDWithString:@"68753A44-4D6F-1
226-9C60-0050E4C00067"]])
        {
            NSLog(@"Service found with UUID: %@", service.UUID);
            //查找服务中的特征值
            [peripheral discoverCharacteristics:nil forService:service];
            break;
        }
    }
}
```

实现发现服务中特征值的回调方法如下所示。

```
//开发服务中的特征值后回调的方法
-(void)peripheral:(CBPeripheral *)peripheral didDiscoverCharacteristicsFo
rService:(CBService *)service error:(NSError *)error{
    for (CBCharacteristic *characteristic in service.characteristics)
    {
        //发现特征 比较特征值得 UUID 来获取所需要的
        if ([characteristic.UUID isEqual:[CBUUID UUIDWithString:@"68753A44
-4D6F-1226-9C60-0050E4C00067"]]) {
            //保存特征值
            _centerReadChara = characteristic;
            //监听特征值
        }
    }
}
```



```

        [_peripheral setNotifyValue:YES forCharacteristic:_centerReadChara];
    }
    if ([characteristic.UUID isEqual:[CBUUID UUIDWithString:@"68753A44-4D6F-1226-9C60-0050E4C00068"]]) {
        //保存特征值
        _centerWriteChara = characteristic;
    }
}
}
}

```

实现所监听的特征值数据更新时的回调方法如下所示。

```

//所监听的特征值更新时回调的方法
- (void)peripheral:(CBPeripheral *)peripheral didUpdateValueForCharacteristic:(CBCharacteristic *)characteristic error:(NSError *)error
{
    //更新接收到的数据
    NSLog(@"%@", [[NSString alloc] initWithData:characteristic.value encoding:NSUTF8StringEncoding]);
    //要在主线程中刷新
    dispatch_async(dispatch_get_main_queue(), ^{
        [self.delegate getData:[[NSString alloc] initWithData:characteristic.value encoding:NSUTF8StringEncoding]];
    });
}

```

除了上面的方法之外，在 `BlueToothTool.m` 文件中还需要实现两个方法，分别为断开连接与向所连接的设备传递数据的方法如下所示。

```

//断开连接
- (void)disconnect{
    if (!_isCentral) {
        [_centerManger cancelPeripheralConnection:_peripheral];
        [_peripheral setNotifyValue:NO forCharacteristic:_centerReadChara];
    }
}
//写数据
- (void)writeData:(NSString *)data{
    if (_isCentral) {
        [_peripheralManager updateValue:[data dataUsingEncoding:NSUTF8StringEncoding] forCharacteristic:_readChara onSubscribedCentrals:nil];
    }else{

```

```

        [_peripheral writeValue:[data dataUsingEncoding:NSUTF8StringEncoding]
forCharacteristic:_centerWriteChara type:CBCharacteristicWriteWithoutResponse];
    }
}

```

7.6.2 棋盘瓦片的设计

本项实战演练的初衷是使读者熟练运用蓝牙技术,在游戏的界面设计采用简易朴素的设计方式,读者如果有兴趣,可以发挥想象进行自定义扩展。

大家都知道,五子棋的棋盘是采用横纵线交错分割而成的,在设计游戏的棋盘时,首先需要设计棋盘上每个可以落子的棋格瓦片,在工程中创建一个新的类,使其继承于 UIButton,命名为 TipButton。在 TipButton.h 文件中声明如下属性。

```

@interface TipButton : UIButton
//标记此瓦片是否已经落子 0 空 1 己方落子 2 敌方落子
@property(nonatomic,assign)int hasChess;
//落子 BOOL 类型的参数 决定是己方还是敌方
-(void)dropChess:(BOOL)isMine;
//设置白子或者黑子
@property(nonatomic,assign)BOOL isWhite;
//瓦片编号
@property(nonatomic,assign)int index;
@end

```

在 TipButton.m 文件中重写初始化方法如下所示。

```

- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        [self creatView];
    }
    return self;
}

```

实现创建视图的方法 creatView 如下所示。

```

-(void)creatView{
    //创建横竖两条线
    UIView * line1 = [[UIView alloc] initWithFrame:CGRectMakeMake(self.frame.size.width/2-0.25, 0, 0.5, self.frame.size.height)];
    line1.backgroundColor = [UIColor grayColor];
    [self addSubview:line1];

    UIView * line2 = [[UIView alloc] initWithFrame:CGRectMakeMake(0, self.frame.

```

```
size.height/2-0.25, self.frame.size.width, 0.5)];
    line2.backgroundColor = [UIColor grayColor];
    [self addSubview:line2];
}
```

实现落子的方法如下所示。

```
-(void)dropChess:(BOOL)isMine{
    UIView * view = [[UIView alloc] initWithFrame:CGRectMake(self.frame.size.width/2-5, self.frame.size.height/2-5, 10, 10)];
    view.layer.masksToBounds = YES;
    view.layer.cornerRadius = 5;
    UIColor * myColor;
    UIColor * otherColor;
    if (_isWhite) {
        myColor = [UIColor whiteColor];
        otherColor = [UIColor blackColor];
    }else{
        myColor = [UIColor blackColor];
        otherColor = [UIColor whiteColor];
    }
    if (isMine) {
        view.backgroundColor = myColor;
        self.hasChess = 1;
    }else{
        view.backgroundColor = otherColor;
        self.hasChess = 2;
    }
    [self addSubview:view];
}
```

7.6.3 核心游戏视图与游戏核心逻辑的设计

TipButton 类用于创建五子棋游戏棋盘上每一个棋格，创建一个继承于 UIView 的类作为游戏的棋盘类，命名为 GameView。GameView 类是五子棋游戏中的界面与逻辑的核心类，这个类将处理交战逻辑，判定胜负逻辑等。在 GameView.h 中导入如下头文件。

```
#import "TipButton.h"
#import "BlueToothTool.h"
```

在 GameView.h 文件中创建一个协议用于处理棋盘落子后的逻辑如下所示。

```
@protocol GameViewDelegate<NSObject>
-(void)gameViewClick:(NSString *)index;
@end
```


在 GameView.h 中声明如下属性。

```
@interface GameView : UIView<UIAlertViewDelegate>
//存放所有棋格
@property(n nonatomic, strong) NSMutableArray<TipButton *> * tipArray;
@property(n nonatomic, weak) id<GameViewDelegate>delegate;
//进行下子
-(void)setTipIndex:(int)index;
@end
```

在 GameView.m 中重写初始化方法如下所示。

```
//创建表格视图 横16 竖20
-(void)createView{
    self.layer.borderColor = [UIColor colorWithRed:222/255.0 green:222/255.0 blue:222/255.0 alpha:1].CGColor;
    self.layer.borderWidth = 0.5;
    CGFloat width = self.frame.size.width/12;
    CGFloat height = self.frame.size.height/20;
    //排列布局
    for (int i=0; i<240; i++) {
        TipButton * btn = [[TipButton alloc] initWithFrame:CGRectMake(width*(i%12), height*(i/12), width, height)];
        [btn addTarget:self action:@selector(click:) forControlEvents:UIControlEventTouchUpInside];
        btn.isWhite = NO;
        btn.index=i;
        [self addSubview:btn];
        [_tipArray addObject:btn];
    }
}
```

实现按钮的触发方法如下所示。

```
-(void)click:(TipButton *)btn{
    if (btn.hasChess==0) {
        //下子
        [btn dropChess:YES];
        //进行胜负判定
        [self check];
        [self.delegate gameViewClick:[NSString stringWithFormat:@"%d", btn.index]];
    }
}
```

实现胜负判定方法 check 如下所示。

```

//进行胜负判定
-(void)cheak{
    //判定己方是否胜利
    if ([self cheakMine]) {
        UIAlertView * alert = [[UIAlertView alloc] initWithTitle:@"您胜利啦"
message:@""" delegate:self cancelButtonTitle:@"好的" otherButtonTitles:nil, nil];
        [alert show];
    }
    if ([self cheakOther]) {
        UIAlertView * alert = [[UIAlertView alloc] initWithTitle:@"您失败了"
message:@""" delegate:self cancelButtonTitle:@"好的" otherButtonTitles:nil, nil];
        [alert show];
    }
}
}

```

游戏结束后会弹出警告框，单击警告框上的按钮后将进行游戏的重置操作，这里采用这样的设计方式：用户单击主界面视图控制器中的触发按钮进入游戏界面视图控制器，当游戏结束后，界面退回主界面视图控制器。实现警告框按钮单击的代理方法如下所示。

```

-(void>alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)
buttonIndex{
    [[BluetoothTool sharedManager]disconnect];
    [(UIViewController *)[self.superview nextResponder] dismissViewContro
llerAnimated:YES completion:nil];
}

```

`cheakMine` 与 `cheakOther` 方法分别用来判定己方是否胜利和对方是否胜利，`cheakMine` 方法实现如下所示。

```

-(BOOL)cheakOther{
    //遍历所有棋子
    for (int i=0; i<_tipArray.count; i++) {
        TipButton * tip = _tipArray[i];
        //获取是否是己方棋子
        if (tip.hasChess==2) {
            //进行五子判定逻辑
            //横向
            if ( [self cheak1HasMineOrOther:NO index:i]) {
                return YES;
            }
            //左上到右下的对角线
            if ( [self cheak2HasMineOrOther:NO index:i]) {
                return YES;
            }
            //纵向
            if ( [self cheak3HasMineOrOther:NO index:i]) {

```



```

        return YES;
    }
    //右上到左下的对角线
    if ( [self cheak4HasMineOrOther:NO index:i]) {
        return YES;
    }
}
}
return NO;
}

```

cheakMine 方法中对己方棋子进行是否连五的判断，是否连五的判断分别通过 4 个方向来进行，横向是否连五，左上到右下的对角线方向是否连五，纵向是否连五，右上到左下的对角线是否连五。4 个方向的连五判定方法实现如下所示。

```

-(BOOL)cheak1HasMineOrOther:(BOOL)mine index:(int)index{
    int mineOrOther = 0;
    if (mine) {
        mineOrOther = 1;
    }else{
        mineOrOther = 2;
    }
    int count=1;
    //左侧右侧同时进行可以增加效率
    //左侧
    count = count +[self algorithmic1:index param:mineOrOther num:4];
    //右侧
    count = count +[self algorithmic2:index param:mineOrOther num:4];
    if (count>=5) {
        return YES;
    }else{
        return NO;
    }
}

-(BOOL)cheak2HasMineOrOther:(BOOL)mine index:(int)index{
    int mineOrOther = 0;
    if (mine) {
        mineOrOther = 1;
    }else{
        mineOrOther = 2;
    }
    int count=1;
    //左上右下同时进行可以增加效率
    //左上
    count = count +[self algorithmic3:index param:mineOrOther num:4];

```



```

//右下
count = count +[self algorithmic4:index param:mineOrOther num:4];
if (count>=5) {
    return YES;
}else{
    return NO;
}
}
-(BOOL)cheak3HasMineOrOther:(BOOL)mine index:(int)index{
    int mineOrOther = 0;
    if (mine) {
        mineOrOther = 1;
    }else{
        mineOrOther = 2;
    }
    int count=1;
    //纵向
    //向上
    count = count +[self algorithmic5:index param:mineOrOther num:4];
    //向下
    count = count +[self algorithmic6:index param:mineOrOther num:4];
    if (count>=5) {
        return YES;
    }else{
        return NO;
    }
}
-(BOOL)cheak4HasMineOrOther:(BOOL)mine index:(int)index{
    int mineOrOther = 0;
    if (mine) {
        mineOrOther = 1;
    }else{
        mineOrOther = 2;
    }
    int count=1;
    //纵向
    //向上
    count = count +[self algorithmic7:index param:mineOrOther num:4];
    //向下
    count = count +[self algorithmic8:index param:mineOrOther num:4];

    NSLog(@"%d",count);
    if (count>=5) {
        return YES;
    }
}

```

```

    }else{
        return NO;
    }
}

```

上面方法中的 `algorithmic:param:mine:num` 系列方法是连五判定的核心算法，其将通过遍历递归的方式返回特定方向所连棋子的个数，算法实现如下所示。

```

/*
左侧递归进行查找 index 棋子编号 param 对比值 num 递归次数
*/
-(int)algorithmic1:(int)index param:(int)param num:(int)num{
    if (num>0) {
        int tem = 4-(num-1);
        //左侧有子
        if (index-tem>=0) {
            //左侧无换行
            if (((index-tem)%12)!=11){
                if (_tipArray[index-tem].hasChess==param) {
                    return [self algorithmic1:index param:param num:num-1];
                }else{
                    return 4-num;
                }
            }else{
                return 4-num;
            }
        }else{
            return 4-num;
        }
    }else{
        //递归了四次
        return 4-num;
    }
}

/*
右侧递归进行查找 index 棋子编号 param 对比值 num 递归次数
*/
-(int)algorithmic2:(int)index param:(int)param num:(int)num{

    if (num>0) {
        int tem = 4-(num-1);
        //右侧有子
        if (index+tem<240) {
            //右侧无换行
            if (((index+tem)%12)!=11){

```



```

        if (_tipArray[index+tem].hasChess==param) {
            return [self algorithmic2:index param:param num:num-1];
        }else{
            return 4-num;
        }
    }else{
        return 4-num;
    }
}
}else{
    return 4-num;
}
}
}else{
    //递归了四次
    return 4-num;
}
}
}
/*
左上递归进行查找 index 棋子编号 param 对比值 num 递归次数
*/
-(int)algorithmic3:(int)index param:(int)param num:(int)num{
    if (num>0) {
        int tem = 4-(num-1);
        //左上子
        if ((index-(tem*12)-tem)>=0) {
            //右侧无换行
            if (((index-(tem*12)-tem)%12)!=11){
                if (_tipArray[(index-(tem*12)-tem)].hasChess==param) {
                    return [self algorithmic3:index param:param num:num-1];
                }else{
                    return 4-num;
                }
            }else{
                return 4-num;
            }
        }else{
            return 4-num;
        }
    }
}
}else{
    //递归了四次
    return 4-num;
}
}
}
//右下方向
-(int)algorithmic4:(int)index param:(int)param num:(int)num{

```



```

        if (num>0) {
            int tem = 4-(num-1);
            //右下子
            if ((index+(tem*12)+tem)<240) {
                //右侧无换行
                if (((index+(tem*12)+tem)%12)!=0){
                    if (_tipArray[(index+(tem*12)+tem)].hasChess==param) {
                        return [self algorithmic4:index param:param num:num-1];
                    }else{
                        return 4-num;
                    }
                }else{
                    return 4-num;
                }
            }else{
                return 4-num;
            }
        }else{
            //递归了四次
            return 4-num;
        }
    }
}
//上方向
-(int)algorithmic5:(int)index param:(int)param num:(int)num{
    if (num>0) {
        int tem = 4-(num-1);
        //上有子
        if ((index-(tem*12))>=0) {
            if (_tipArray[(index-(tem*12))].hasChess==param) {
                return [self algorithmic5:index param:param num:num-1];
            }else{
                return 4-num;
            }
        }else{
            return 4-num;
        }
    }else{
        //递归了四次
        return 4-num;
    }
}
}
//下方向
-(int)algorithmic6:(int)index param:(int)param num:(int)num{
    if (num>0) {

```

```

int tem = 4-(num-1);
//下有子
if ((index+(tem*12))<240) {
    if (_tipArray[(index+(tem*12))].hasChess==param) {
        return [self algorithmic6:index param:param num:num-1];
    }else{
        return 4-num;
    }
}
}else{
    return 4-num;
}
}
}else{
    //递归了四次
    return 4-num;
}
}

//右上放下
-(int)algorithmic7:(int)index param:(int)param num:(int)num{
    if (num>0) {
        int tem = 4-(num-1);
        //右上有子
        if ((index-(tem*12)+tem)>=0) {
            //右侧无换行
            if (((index-(tem*12)+tem)%12)!=0) {
                if (_tipArray[(index-(tem*12)+tem)].hasChess==param) {
                    return [self algorithmic7:index param:param num:num-1];
                }else{
                    return 4-num;
                }
            }
        }else{
            return 4-num;
        }
    }
    }else{
        return 4-num;
    }
}
}else{
    //递归了四次
    return 4-num;
}
}

//左下方向
-(int)algorithmic8:(int)index param:(int)param num:(int)num{
    if (num>0) {
        int tem = 4-(num-1);

```



```

//左下有子
if ((index+(tem*12)-tem)<240) {
    //左侧无换行
    if (((index+(tem*12)-tem)%12)!=11){
        if (_tipArray[(index+(tem*12)-tem)].hasChess==param) {
            return [self algorithmic8:index param:param num:num-1];
        }else{
            return 4-num;
        }
    }else{
        return 4-num;
    }
}
}else{
    return 4-num;
}
}
//递归了四次
return 4-num;
}
}

```

实现 `cheakOther` 方法如下所示。

```

-(BOOL)cheakOther{
    //遍历所有棋子
    for (int i=0; i<_tipArray.count; i++) {
        TipButton * tip = _tipArray[i];
        //获取是否是对方棋子
        if (tip.hasChess==2) {
            //进行五子判定逻辑
            //横向
            if ([self cheak1HasMineOrOther:NO index:i]) {
                return YES;
            }
            //左上到右下的对角线
            if ([self cheak2HasMineOrOther:NO index:i]) {
                return YES;
            }
            //纵向
            if ([self cheak3HasMineOrOther:NO index:i]) {
                return YES;
            }
            //右上到左下的对角线
            if ([self cheak4HasMineOrOther:NO index:i]) {
                return YES;
            }
        }
    }
}

```



```

    }
}
return NO;
}

```

最后，实现落子的方法 `setTipIndex` 如下所示。

```

-(void)setTipIndex:(int)index{
    //下子
    for (TipButton * btn in _tipArray) {
        if (btn.index==index) {
            [btn dropChess:NO];
            [self check];
        }
    }
}

```

7.6.4 核心游戏视图控制器的设计

创建一个继承于 `UIViewController` 的类，命名为 `GameViewController`，作为游戏的核心视图控制器。在 `GameViewController.m` 中导入如下头文件。

```

#import "GameView.h"
#import "BluetoothTool.h"

```

在 `GameViewController.m` 文件中遵守协议并声明相关属性如下所示。

```

@interface GameViewController ()<BluetoothToolDelegate,GameViewDelegate>
{
    UIView * _bgView;
    UILabel * _tipLabel;
    GameView * _view;
}
@end

```

在 `viewDidLoad` 方法中编写如下初始化代码。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    self.view.backgroundColor = [UIColor brownColor];
    //创建游戏视图
    _view = [[GameView alloc] initWithFrame:CGRectMake(20, 40, (self.view.frame.size.width-40), (self.view.frame.size.width-40)/12*20)];
    _view.delegate=self;
    [self.view addSubview:_view];
}

```

```

//创建背景视图
_bgView = [[UIView alloc] initWithFrame:self.view.frame];
_bgView.backgroundColor = [UIColor colorWithRed:1 green:1 blue:1 alpha:
0.1];

UIButton * btn = [UIButton buttonWithType:UIButtonTypeSystem];
btn.frame = CGRectMake(self.view.frame.size.width/2-50, 150, 100, 30);
UIButton * btn2 = [UIButton buttonWithType:UIButtonTypeSystem];
btn2.frame = CGRectMake(self.view.frame.size.width/2-50, 250, 100, 30);
[btn setTitle:@"创建游戏" forState:UIControlStateNormal];
[btn2 setTitle:@"扫描附近游戏" forState:UIControlStateNormal];
btn.backgroundColor = [UIColor orangeColor];
btn2.backgroundColor = [UIColor orangeColor];
[btn addTarget:self action:@selector(creatGame) forControlEvents:UICo
ntrolEventTouchUpInside];
[btn2 addTarget:self action:@selector(searchGame) forControlEvents:UI
ControlEventTouchUpInside];
[_bgView addSubview:btn];
[_bgView addSubview:btn2];

[self.view addSubview:_bgView];
//设置蓝牙通讯类代理
[BluetoothTool sharedManager].delegate=self;
//创建提示标签
_tipLabel = [[UILabel alloc] initWithFrame:CGRectMake(0, 0, self.view.f
rame.size.width,40)];
[self.view addSubview:_tipLabel];
_tipLabel.textAlignment = NSTextAlignmentCenter;
}

```

实现按钮的触发方法如下所示。

```

-(void)creatGame{
    [[BluetoothTool sharedManager]setUpGame:@" " block:^(BOOL first) {
        [_bgView removeFromSuperview];
        if (first) {
            _tipLabel.text = @"请您下子";
            //进行发送下子信息
        }else{
            _tipLabel.text = @"请等待对方下子";
            self.view.userInteractionEnabled = NO;
            [self gameViewClick:@"-1"];
        }
    }];
}

```

```

-(void)searchGame{
    [[BluetoothTool sharedManager]searchGame];
}

```

实现相应代理方法如下所示。

```

-(void)getData:(NSString *)data{
    if (_bgView.superview) {
        [_bgView removeFromSuperview];
    }
    if ([data integerValue]==-1) {
        _tipLabel.text = @"请您下子";
        self.view.userInteractionEnabled = YES;
        return;
    }
    _tipLabel.text = @"请您下子";
    [_view setTipIndex:[data intValue]];
    self.view.userInteractionEnabled = YES;
}

-(void)gameViewClick:(NSString *)index{
    _tipLabel.text = @"请等待对方下子";
    [[BluetoothTool sharedManager]writeData:index];
    self.view.userInteractionEnabled = NO;
}

```

到此，蓝牙五子棋游戏的核心代码全部编写完毕，在系统工程模板中的 ViewController 类中拉入一个标签，提示用户单击屏幕即可开始游戏，Main.storyboard 中模样如图 7-24 所示。

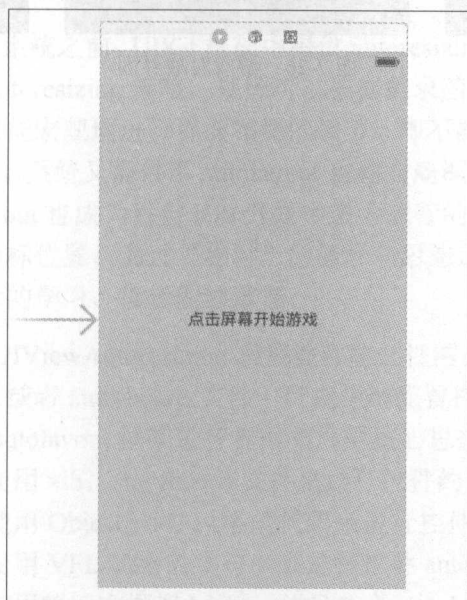


图 7-24 向初始化视图控制器中添加一行文字

向 ViewController.m 文件中引入 GameViewController 的头文件如下所示。

```
#import "GameViewController.h"
```

在 ViewController.m 文件中实现 touchesBegan:withEvent: 方法如下所示。

```
-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{  
    [self presentViewController:[[GameViewController alloc] init] animated:  
    YES completion:nil];  
}
```

游戏需要在两台设备上运行工程，游戏的几个核心界面效果如图 7-25~图 7-27 所示。

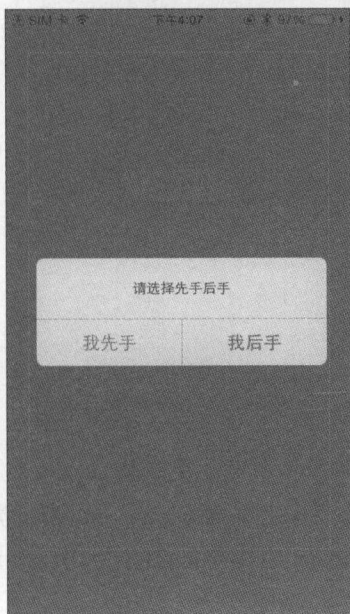


图 7-25 开始游戏界面

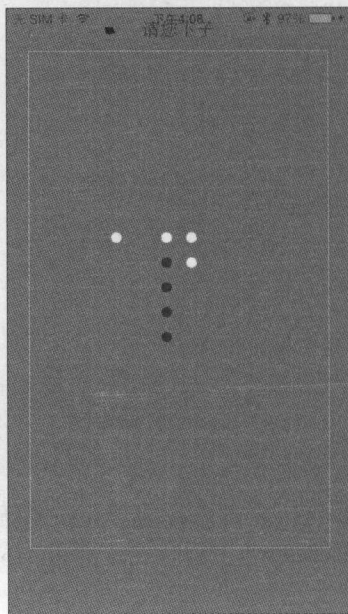


图 7-26 游戏对战界面

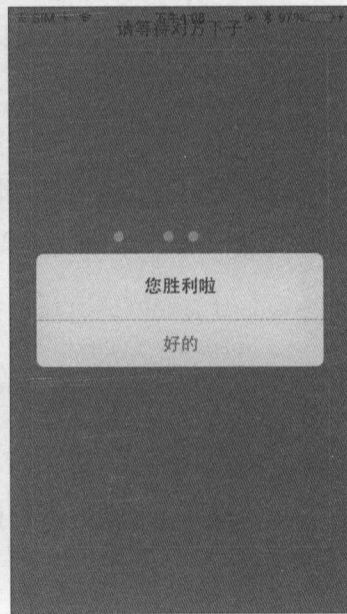


图 7-27 游戏结束界面

第 8 章

界面布局

iOS 界面的开发离不开坐标系，前面向读者介绍的所有 UI 控件，都是通过 `frame` 或者 `center` 属性来设置其在父视图上的位置的。这种通过精确坐标的方式进行控件位置的设置方法有一个很大的缺陷，可能开发者在所参照的屏幕上布局没有问题，但布局样式却在换了不同屏幕尺寸的设备上完全错乱。并且，随着 iOS 设备越来越多样化，在设计 iOS 程序时，界面布局策略也越来越重要。

在 iOS 6 系统之前，UIKit 框架中提供 `autoresizing` 方式进行简单的视图自适应，通过 `autoresizing` 策略，视图可以根据需求随父视图的变化而变化，然而这种方式只能宏观地进行界面布局的调节，却不能精确地处理布局的细节，在 iOS 6 之后，系统又提供了 `autolayout` 自动布局框架来解决布局适配的相关问题，`autolayout` 也成为目前 iOS 开发中最为流行的界面布局方式，它解放了控件的具体坐标位置，通过“相对”的概念与思想进行界面的布局管理。

通过本章的学习，读者能够掌握：

1. 使用 `UIViewAutoresizing` 设置控件随父视图变化而变化的方式。
2. 在 `xib` 或者 `storyboard` 文件中可视化地配置控件的 `autoresizing` 属性。
3. 使用 `autolayout` 框架进行界面布局的核心思想。
4. 学会使用 `xib`、`storyboard` 文件来进行控件约束的添加。
5. 学会使用 Objective-C 风格的代码来进行控件 `autolayout` 约束的设置。
6. 学会使用 VFL 风格的字符串来进行控件 `autolayout` 约束的设置。
7. 学会使用第三方框架 `Masonry` 进行控件 `autolayout` 约束的设置。

8.1 iOS 中传统的 UIViewAutoresizing 布局模式

UIViewAutoresizing 是 iOS 开发中一种传统的自动布局方法，它主要的作用是设置当视图控件的父视图尺寸进行改变时子视图进行相应尺寸与位置的调整。UIViewAutoresizing 实际上是 UIKit 框架中的一个枚举，其枚举值与意义如下所示。

```
typedef NS_OPTIONS(NSUInteger, UIViewAutoresizing) {
    UIViewAutoresizingNone                = 0, // 默认
    UIViewAutoresizingFlexibleLeftMargin  = 1 << 0, // 与父视图右边间距固定，左
边可变
    UIViewAutoresizingFlexibleWidth      = 1 << 1, // 视图宽度可变
    UIViewAutoresizingFlexibleRightMargin = 1 << 2, // 与父视图左边间距固定，右
边可变
    UIViewAutoresizingFlexibleTopMargin   = 1 << 3, // 与父视图下边间距固定，上
边可变
    UIViewAutoresizingFlexibleHeight     = 1 << 4, // 视图高度可变
    UIViewAutoresizingFlexibleBottomMargin = 1 << 5, // 与父视图上边间距固定，下边
可变
};
```

上面的枚举约定了子视图随父视图变化而调整的规则。



提示

由 NS_OPTIONS 可以得知，UIViewAutoresizing 枚举中的枚举值是可以进行按位或运算来结合使用的。

8.1.1 通过代码来设置视图控件的 UIViewAutoresizing 模式

使用 Xcode 创建一个名为 UIViewAutoresizingTest 的工程。在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码来做演示。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIView * view1 = [[UIView alloc] initWithFrame:CGRectMake(20, 40, 200,
200)];
    view1.backgroundColor=[UIColor redColor];
    UIView * view2 = [[UIView alloc] initWithFrame:CGRectMake(10, 10, 100,
100)];
    view2.backgroundColor=[UIColor greenColor];
    [view1 addSubview:view2];
    [self.view addSubview:view1];
}
```


上面代码创建了两个大小不同的色块,将绿色的小色块作为子视图添加在了红色的大色块上面。此时运行工程,效果如图 8-1 所示。

添加如下代码设置绿色色块的自适应模式并修改红色大色块的尺寸,代码如下所示。

```
view2.autoresizingMask = UIViewAutoresizingFlexibleLeftMargin;
view1.frame = CGRectMake(20, 40, 300, 300);
```

再次运行工程,效果如图 8-2 所示。

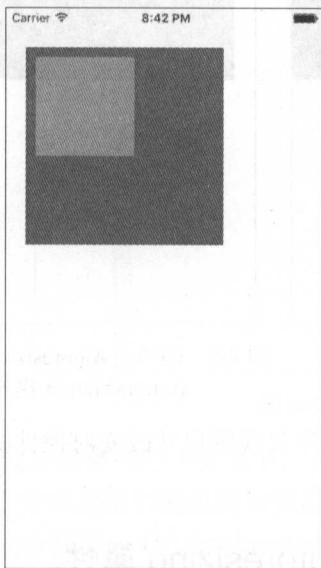


图 8-1 在视图中添加两个色块

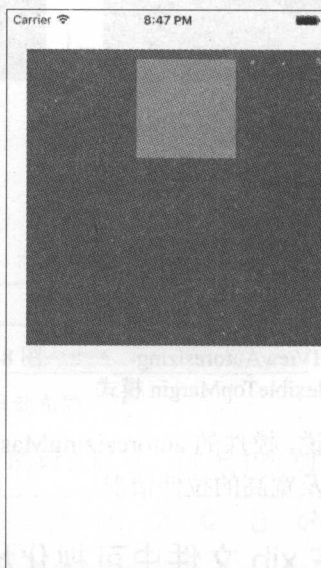


图 8-2 UIViewAutoresizingFlexibleLeftMargin 模式

由图 8-2 可以看出, UIViewAutoresizingFlexibleLeftMargin 模式中,当父视图尺寸改变时,子视图右边侧与父视图右边侧距离保持不变。其他各种自适应模式的效果如图 8-3~图 8-7 所示。

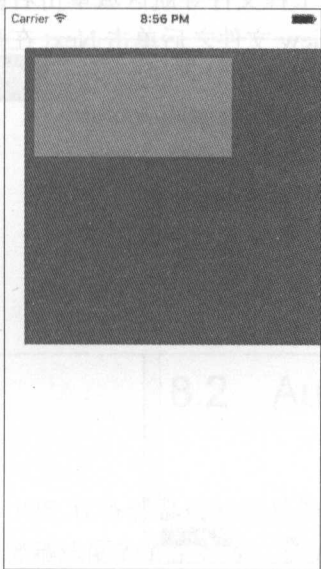


图 8-3 UIViewAutoresizingFlexibleWidth 模式

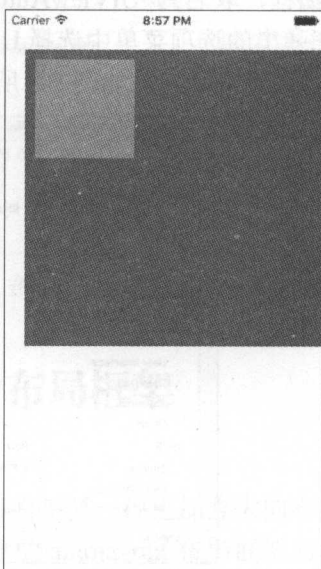


图 8-4 UIViewAutoresizingFlexibleRightMargin 模式

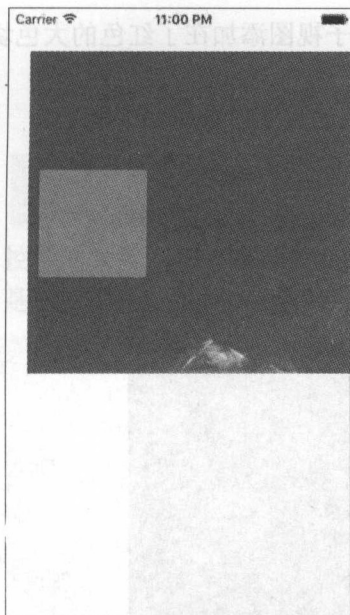


图 8-5 UIViewAutoresizingFlexibleTopMargin 模式

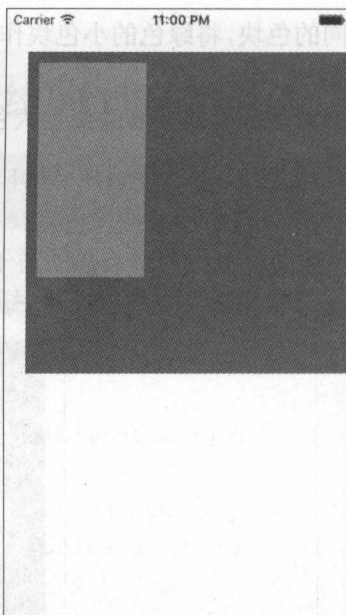


图 8-6 UIViewAutoresizingFlexibleHeight 模式

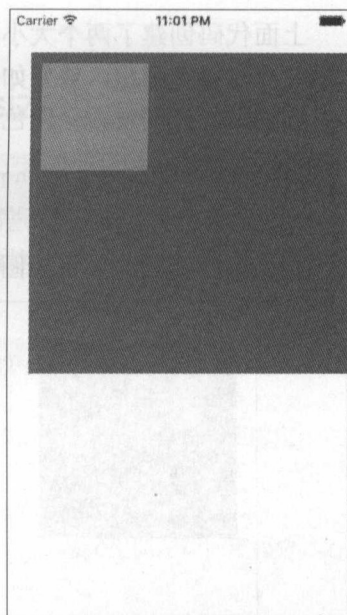


图 8-7 UIViewAutoresizingFlexibleBottomMargin 模式

简单来说, 控件的 `autoresizingMask` 属性就是设置当控件父视图尺寸改变时控件的上、下、左、右边距及宽高的拉伸情况。

8.1.2 在 xib 文件中可视化地配置控件的 autoresizing 属性

开发者也可以在 xib 可视化视图控件编辑文件中配置视图控件的 `autoresizing` 属性。使用 Xcode 在创建某些特定类时, 开发者可以自由选择是否一并创建 xib 关联文件。使用 Xcode 创建一个新的工程, 取名为 `UIViewAutoresizingXIBTest`。在工程文件导航区域单击右键, 选择 `New File`, 在弹出的选项菜单中选择 `User Interface`, 选择 `View` 文件之后单击 `Next` 在弹出的菜单中输入文件名完成创建, 如图 8-8 所示。

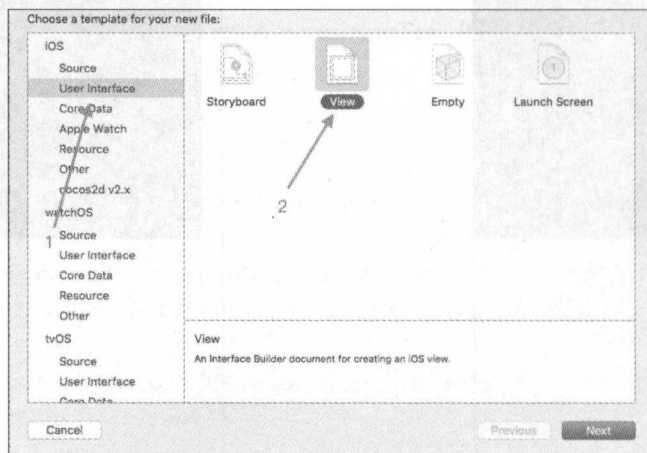


图 8-8 创建 View 视图 xib 文件

xib 文件与先前使用的 storyboard 文件有些类似，都是一种可视化的开发工具，上面创建好的 xib 中自带一个 UIView 视图，Xcode 默认是使用 autolayout 模式进行自动布局的，开发者若需要使用 autoresizing 进行布局控制，需要先关闭 autolayout，选中 xib 文件中的 View 视图，在右侧的工具栏中找到 Use Auto Layout 选项，将勾选去掉，如图 8-9 所示。

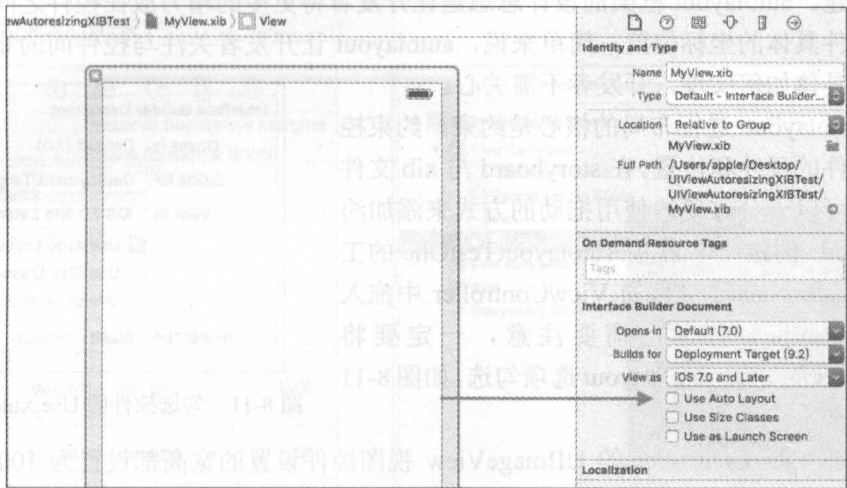


图 8-9 取消使用 autolayout 自动布局

在右侧工具栏的尺寸标签下，可以找到 autoresizing 的设置区域，区域中分别有上、下、左、右 4 个方向边距的固定线与宽度高度指示线，这些线分别对应 UIViewAutoresizing 枚举中的相关值，选中上、下、左、右边距线代表相应的边距固定不变，选中宽度或者高度指示线代表控件的宽度或者高度可拉伸，如图 8-10 所示。

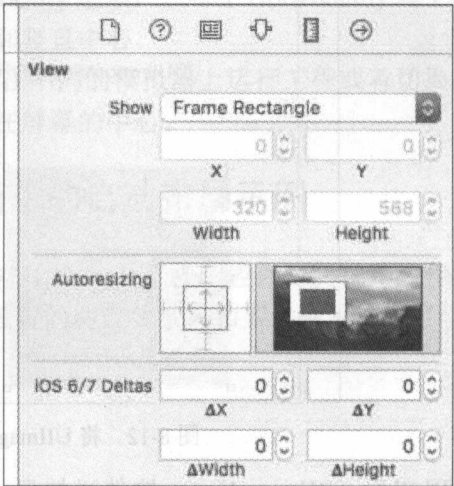



图 8-10 在 xib 文件中设置控件的 autoresizing 属性

**提示**

在 xib 文件进行视图控件的 autoresizing 属性的设置时，选中相应的指示线后，右侧的预览区域会以动画的效果展示视图的自适应效果。开发者可利用这一特性进行调试。

8.2 Autolayout 自动布局框架

随着 iOS 设备屏幕尺寸型号的多样化，开发者迫切的需要一种更加强大的布局框架来支持 iOS 多屏幕适配的界面开发。在 iOS 6 之后，系统提供的 autolayout 布局框架为 iOS 的界面布局增添了一种新的方法与思路。

8.2.1 初识 autolayout

正如 storyboard 文件的设计目的是让开发者可以将更多的精力放在编程的业务逻辑上而不是界面搭建。autolayout 框架的设计思想是让开发者将更多的精力放在控件之间的位置关系上而不是控件具体的坐标位置。简单来说，autolayout 让开发者关注与控件间的布局关系，至于这些关系具体如何实现，开发者不需关心。

使用 autolayout 进行布局的核心是约束。约束控制与调节控件的尺寸和位置。在 storyboard 与 xib 文件中，开发者可以十分方便的使用拖动的方式来添加约束。使用 Xcode 创建一个名为 AutolayoutTestOne 的工程，在 main.storyboard 文件的 ViewController 中拖入一个 UIImageView 控件，需要注意，一定要将 ViewController 的 Use Auto Layout 选项勾选，如图 8-11 所示。

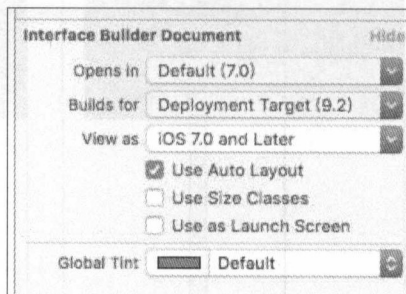


图 8-11 勾选控件的 Use Auto Layout 选项

将拖动进 ViewController 的 UIImageView 视图控件设置的宽高都设置为 100，如图 8-12 所示。

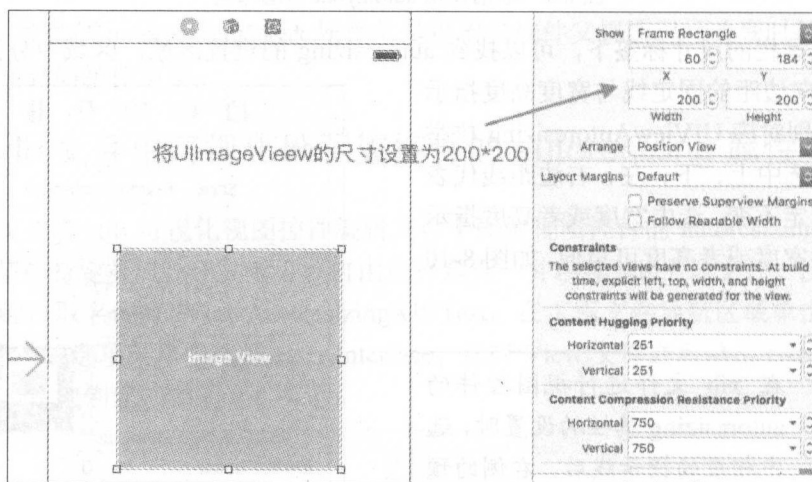


图 8-12 将 UIImageView 控件尺寸设置为 200*200

现在为 UIImageView 控件添加尺寸的约束，鼠标选中 UIImageView 控件，按住 control 键不放，将鼠标拖动至 UIImageView 本身，会弹出如图 8-13 所示的菜单。

将弹出菜单中的 Width 与 Height 选项选中，Width 选项的意义为约束控件的宽度始终保持不变，Height 选项的意义为约束控件的高度始终保持不变。设置这两个约束后，无论设备屏幕尺寸如何变化，UIImageView 控件的尺寸始终确定。

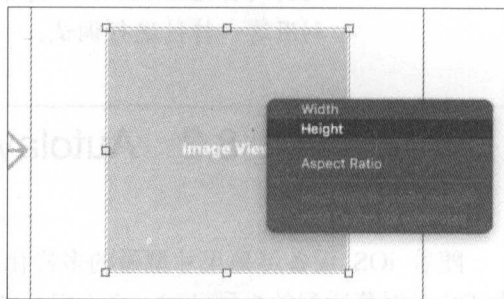


图 8-13 添加约束菜单

在 Xcode 右侧的工具栏中可以查看控件所添加的约束情况, 如图 8-14 所示。

现在再为 UIImageView 控件添加位置的约束, 例如将其约束在屏幕的正中间。依然可以使用类似上面约束控件宽高的方法来约束位置, 也可以在 storyboard 文件中左侧的文件导航区选中 UIImageView 控件, 按住 control 键不放将鼠标移动至其父视图上, 这是会弹出添加约束的菜单, 如图 8-15 所示。

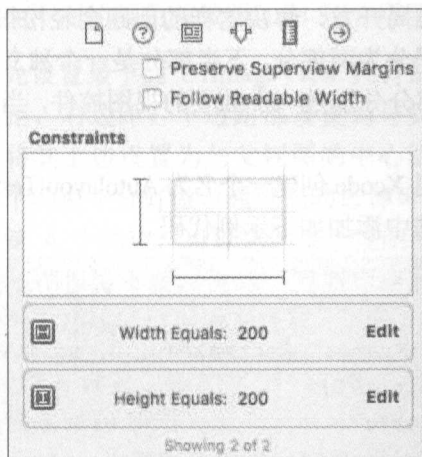


图 8-14 在 Xcode 的工具栏中查看控件添加的约束

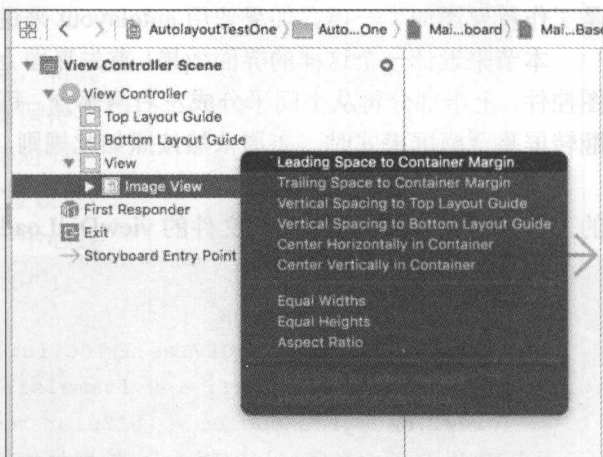


图 8-15 为视图控件添加约束

选中菜单中的 Center Horizontally 和 Center Vertically 选项, Center Horizontally 与 Center Vertically 分别会将控件约束在其父视图的水平中心和竖直中心。

将 UIImageView 的背景颜色设置为红色, 此时在不同的模拟器上运行工程或者切换模拟器的横竖屏模式, 会发现 UIImageView 始终会调整在屏幕的中心。

8.2.2 autolayout 的属性意义与一个简单的自动布局示例

使用 autolayout 进行约束布局时, 主要有两种类型, 一种是子视图控件与父视图控件之间的约束关系, 一种是平级视图之间的约束关系。可设置的约束关系有如下几种。

- Width: 对视图宽度的约束
- Height: 对视图高度的约束
- Aspect Ratio :对视图的宽高比进行约束
- Leading Space to Container Margin: 对视图左边距与其他视图进行约束
- Trailing Space to Container Margin: 对视图右边距与其他视图进行约束
- Vertical Spacing to Top Layout Guide: 对视图上边距与其他视图进行约束
- Vertical Spacing to Bottom Layout Guide: 对视图下边距与其他视图进行约束
- Center Horizontally in Container: 对视图竖直中心线与其他视图进行约束
- Center Vertically in Container: 对视图水平中心线与其他视图进行约束
- Equal Widths: 约束两视图宽度相等
- Equal Heights: 约束两视图高度相等

上面介绍的这些约束方式是 Autolayout 中进行控件布局约束和核心方法,下面将通过演示一个经典的 autolayout 布局示例来使读者更加深入的体会 Autolayout 布局的强大之处。

在开发中经常会遇到需要支持横屏显示的界面,每当这种情况,开发者一定会遇到一个问题,在竖屏模式下排好的布局,屏幕一横就错乱了,适配了横屏,竖屏模式又出了问题,这是令人十分头疼的问题。根据屏幕状态来写两套布局方案是可以解决这个问题,可是更大的问题是工作量要增加了一倍。如果使用 autolayout 来进行布局开发,解决这样的问题会轻松很多。

本节来设计一个这样的界面效果:首先界面上下平分为两部分,下半部分是一个独立的视图控件,上半部分再从中间平分成左右两部分,每一部分分别是一个独立的视图控件,当用户翻转屏幕至横屏模式时,布局依然按照如此规则。

先来以之前的布局方法来完成这个需求试试看,使用 Xcode 创建一个名为 AutolayoutTestTwo 的工程,在 ViewController.m 文件的 viewDidLoad 方法中添加如下示例代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIView * view1 = [[UIView alloc] initWithFrame:CGRectMake(0, 0, self.view.frame.size.width/2, self.view.frame.size.height/2)];
    view1.backgroundColor = [UIColor redColor];
    UIView * view2 = [[UIView alloc] initWithFrame:CGRectMake(self.view.frame.size.width/2, 0, self.view.frame.size.width/2, self.view.frame.size.height/2)];
    view2.backgroundColor = [UIColor blueColor];
    UIView * view3 = [[UIView alloc] initWithFrame:CGRectMake(0, self.view.frame.size.height/2, self.view.frame.size.width, self.view.frame.size.height/2)];
    view3.backgroundColor = [UIColor greenColor];
    [self.view addSubview:view1];
    [self.view addSubview:view2];
    [self.view addSubview:view3];
}
```

运行项目,可以看到竖屏和横屏的效果分别如图 8-16 与图 8-17 所示。

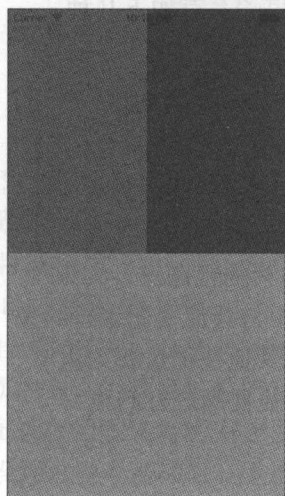


图 8-16 竖屏模式下的三色块布局



图 8-17 横屏模式下的 3 色块布局

从图 8-16 与图 8-17 中可以看出,横屏模式下视图的布局完全错乱。将 viewDidLoad 中的代码删掉,下面在 storyboard 中使用 autolayout 来进行三色块视图的布局。首先向 main.storyboard 文件中的 ViewController 中拖入 3 个 UIImageView 对象,将其设置为不同的颜色并按照如图 8-18 所示摆放。

先设置最下面 UIImageView 控件的约束,将其左边距,右边距,下边距都设置为与父视图保持不变,将其上边设置为与父视图的中心线保持对齐,前 3 个约束十分好做,通过拖动选择可以直接设置,对于第 4 个约束,可以先约束子视图的水平中心线与父视图保持不变,再将子视图约束的水平中心线修改为上边线。过程如下所示。

(1) 设置视图的水平中心线与父视图保持不变,如图 8-19 所示。

(2) 单击视图控件,在 Xcode 右侧工具栏中找到相应约束,双击进入约束设置界面,如图 8-20 所示。

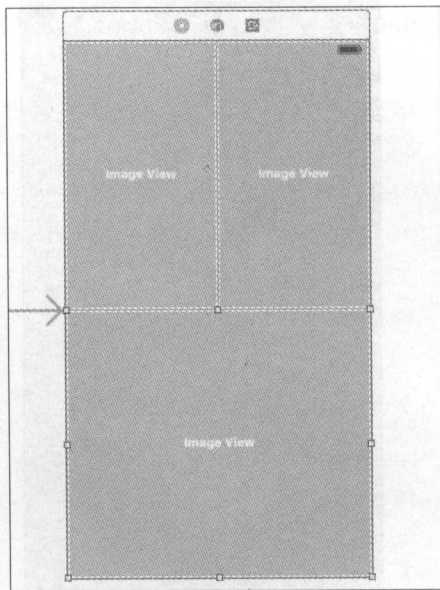


图 8-18 向 ViewController 中拉入 3 个色块

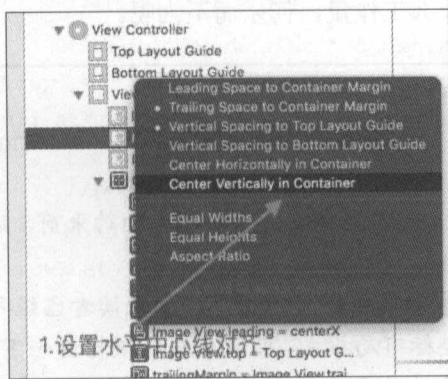


图 8-19 设置视图中心线约束

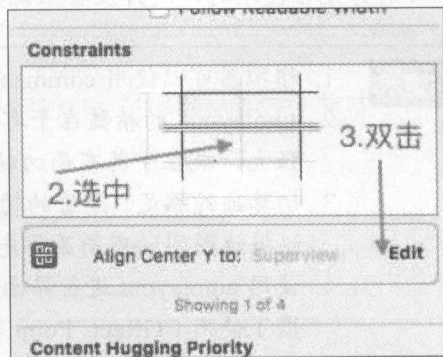


图 8-20 进行约束设置界面

(3) 在约束设置界面将子视图控件的约束水平中心线修改为上边线,如图 8-21 所示。

以同样的方法约束左上角的视图左边,上边与父视图左边,上边距离保持不变,右边与父视图竖直中心线保持对齐,下边与父视图水平中心线保持对齐。约束右上角的视图右边,上边与父视图右边,上边保持不变,左边与父视图竖直中心线保持对齐,下边与父视图水平中心线保持对齐。

再次运行项目,竖屏和横屏的界面布局效果如图 8-22 与图 8-23 所示。

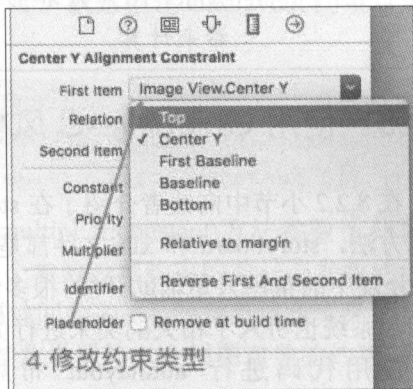


图 8-21 修改约束类型

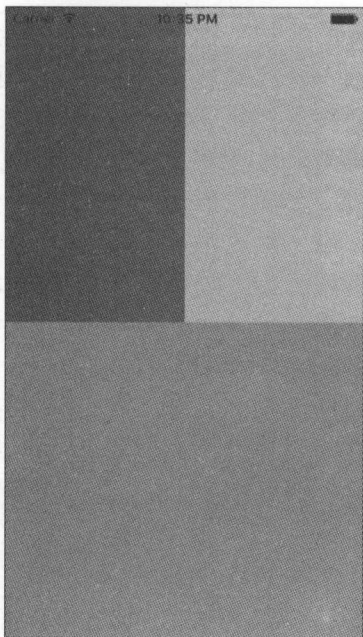


图 8-22 竖屏模式下的 autolayout 布局界面

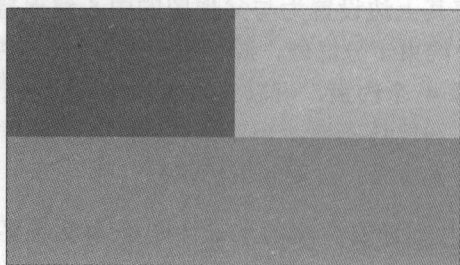


图 8-23 横屏模式下的 autolayout 布局界面

可以发现，使用 Autolayout 进行约束布局，很好地解决了前面提出的需求问题，并且和 storyboard 进行结合使用时，对开发者来说基本没有什么工作量，何乐而不为呢。

提示

1. 模拟器可以使用 `command+左右键` 的方式来进行翻转切换。
2. autolayout 的精髓在于有足够多的约束，autolayout 之所以比 `autoresizing` 强大，就在于其布局的精确性，而精确性正是由约束来提供的。
3. 切莫画蛇添足，矛盾的约束会使 Xcode 发生混乱，所以在添加约束前，建议将试图间的布局关系先整理出来。
4. 使用 autolayout 进行界面布局，首先应该转变的是思路，如果读者已经习惯了使用 `CGRect`、`Point` 等传统的坐标布局模式，那么应该稍微转变一下，Autolayout 倡导的是一个相对的概念，你需要将更多的关注放在视图间的关系，比如 A 和 B 距离 10，A 和 C 右对齐等。具体的坐标会有 autolayout 帮来计算。

8.2.3 使用 Objective-C 风格的方法进行代码 autolayout 布局

在 8.2.2 小节中向读者介绍了在 storyboard 和 xib 文件中使用 autolayout 进行界面布局的思路和方法，storyboard 和 xib 文件都是帮助开发者进行界面开发搭建的可视化编程工具，其在 iOS 开发中的角色只是辅助开发，很多情况下项目中需要的 UI 控件都需要用代码来写，在 iOS6 之后，系统也引入了相关的类来进行 Autolayout 的代码方式创建与布局设置。

使用代码进行 autolayout 布局首先要了解一个重要的类：`NSLayoutConstraint`。`NSLayoutConstraint` 类是进行代码 autolayout 布局的核心类，其创建出具体的自动布局约束对象。

使用 Xcode 创建一个名为 AutolayoutTestThree 的工程, 在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIView * view1 = [[UIView alloc] init];
    view1.translatesAutoresizingMaskIntoConstraints = NO;
    view1.backgroundColor = [UIColor redColor];
    //添加约束前 必须将子视图添加在父视图上
    [self.view addSubview:view1];
    //将 view1 视图约束在屏幕垂直方向的中间
    NSLayoutConstraint * constraintX = [NSLayoutConstraint constraintWithItem:view1 attribute:NSLayoutAttributeCenterX relatedBy:NSLayoutRelationEqual toItem:self.view attribute:NSLayoutAttributeCenterX multiplier:1 constant:0];
    [self.view addConstraint:constraintX];
    //将 view1 视图约束在屏幕水平方向的中间
    NSLayoutConstraint * constraintY = [NSLayoutConstraint constraintWithItem:view1 attribute:NSLayoutAttributeCenterY relatedBy:NSLayoutRelationEqual toItem:self.view attribute:NSLayoutAttributeCenterY multiplier:1 constant:0];
    [self.view addConstraint:constraintY];
    //将视图的宽度约束为 100
    NSLayoutConstraint * constraintW = [NSLayoutConstraint constraintWithItem:view1 attribute:NSLayoutAttributeWidth relatedBy:NSLayoutRelationEqual toItem:nil attribute:NSLayoutAttributeNotAnAttribute multiplier:1 constant:100];
    [view1 addConstraint:constraintW];
    //将视图的高度约束为 100
    NSLayoutConstraint * constraintH = [NSLayoutConstraint constraintWithItem:view1 attribute:NSLayoutAttributeHeight relatedBy:NSLayoutRelationEqual toItem:nil attribute:NSLayoutAttributeNotAnAttribute multiplier:1 constant:100];
    [view1 addConstraint:constraintH];
}
```

上面的代码虽然繁杂, 然而运用到的核心方法只有一个, 就是创建 autolayout 约束对象的如下方法。

```
NSLayoutConstraint * constraintX = [NSLayoutConstraint constraintWithItem:
view1 attribute:NSLayoutAttributeCenterX relatedBy:NSLayoutRelationEqual toItem:
self.view attribute:NSLayoutAttributeCenterX multiplier:1 constant:0];
```

constraintWithItem:attribute:relatedBy:toItem:attribute:multiplier:constant:方法参数十分多, 有 7 个。第 1 个参数设置要约束的第一个视图对象, 第 2 个参数设置约束的第一个参数的约束属性, 具体参数意义后面会介绍。第 3 个参数设置约束属性间的关系, 参数具体意义后面会介绍。第 4 个参数设置要约束的第二个视图对象。第 5 个参数设置第二个视图对象的约束属性。第 6 个参数设置约束的比例。第 7 个参数设置约束的值。

创建约束对象方法中的第 2 个参数和第 5 个参数都需要设置为 `NSLayoutAttribute` 类型的枚举，设置枚举设置的值确定要约束控件的具体属性，常用枚举值及意义如下所示。

```
typedef NS_ENUM(NSInteger, NSLayoutAttribute) {
    NSLayoutAttributeLeft = 1,    //约束控件的左边
    NSLayoutAttributeRight,       //约束控件的右边
    NSLayoutAttributeTop,         //约束控件的顶边
    NSLayoutAttributeBottom,      //约束控件的底边
    NSLayoutAttributeLeading,      //约束控件的前边
    NSLayoutAttributeTrailing,    //约束控件的后边
    NSLayoutAttributeWidth,       //约束控件的宽度
    NSLayoutAttributeHeight,      //约束控件的高度
    NSLayoutAttributeCenterX,     //约束控件中心横轴位置
    NSLayoutAttributeCenterY,     //约束控件中心纵轴位置
    NSLayoutAttributeBaseline,    //约束带文字控件的文字基线位置
    NSLayoutAttributeNotAnAttribute = 0 //缺省属性
};
```

上面的枚举值中，在从左向右的布局结构中，`NSLayoutAttributeLeft`、`NSLayoutAttributeRight` 和 `NSLayoutAttributeLeading`、`NSLayoutAttributeTrailing` 效果一致。上面所有枚举值中，有一个比较特殊，就是 `NSLayoutAttributeNotAnAttribute` 值没有任何意义，只为作为某些情况下方法中的占位，例如下面所示。

```
NSLayoutConstraint * constraintH = [NSLayoutConstraint constraintWithItem:
view1 attribute:NSLayoutAttributeHeight relatedBy:NSLayoutRelationEqual toItem:
m:nil attribute:NSLayoutAttributeNotAnAttribute multiplier:1 constant:100];
```

创建约束对象的方法中第 3 个参数需要设置为 `NSLayoutRelation` 类型的枚举，这个值决定所约束属性间的关系，枚举值及意义如下所示。

```
typedef NS_ENUM(NSInteger, NSLayoutRelation) {
    NSLayoutRelationLessThanOrEqualTo = -1, //小于等于所约束的值
    NSLayoutRelationEqual = 0,              //严格等于所约束的值
    NSLayoutRelationGreaterThanEqual = 1,   //大于等于所约束的值
};
```

创建约束对象方法的最后两个参数决定约束值，第 6 个参数设置约束的比例，第 7 个设置具体的约束值。例如需要设置第 1 个控件的宽度是第 2 个控件宽度的 2 倍还多 100 个单位距离，使用如下代码。

```
NSLayoutConstraint * constraintW = [NSLayoutConstraint constraintWithItem:
view1 attribute:NSLayoutAttributeWidth relatedBy:NSLayoutRelationEqual toItem:
view2 attribute:NSLayoutAttributeNotAnAttribute multiplier:2 constant:100];
```

`multiplier` 与 `constant` 参数的计算方法遵守如下公式。

```
view1 宽度 = view2 宽度 * multiplier + constant
```

运行工程, 可以看到, 无论横屏竖屏模式, 也无论设置屏幕尺寸, 色块控件始终出现在屏幕的正中央, 宽度高度为 100 个单位。



提示

1. 在使用 `addConstraint:` 方法添加约束之前, 必须先将子视图添加到父视图上, 否则会出错。
2. 某些与父视图相关的约束, 约束对象必须添加在父视图上。
3. 若要使用代码添加控件的 `Autolayout` 约束, 必须先将控件的 `translatesAutoresizingMaskIntoConstraints` 属性设置为 `NO`。

8.2.4 使用格式化的字符串进行 autolayout 布局对象的创建

使用 8.2.3 小节介绍的方法进行代码 `autolayout` 布局时, 有一个致命的缺陷, 一个十分简单的布局结构却要写十分冗长的代码。这不仅不利于代码的简洁性, 在调试代码时也给开发者带来很大的麻烦。开发 `autolayout` 的工程师们考虑到了这点, 他们艺术性的创造了一种象形的方式进行 `autolayout` 约束对象的创建。

使用 Xcode 创建一个名为 `AutolayoutTestFour` 的工程, 在 `ViewController.m` 文件的 `viewDidLoad` 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIView * view = [[UIView alloc] init];
    view.translatesAutoresizingMaskIntoConstraints = NO;
    view.backgroundColor = [UIColor redColor];
    [self.view addSubview:view];
    NSArray * constraintArray = [NSLayoutConstraint constraintsWithVisualFormat:@"H:|-20-[view(100@1000)]" options:0 metrics:nil views:NSDictionaryOfVariableBindings(view)];
    NSArray * constraintArray2 = [NSLayoutConstraint constraintsWithVisualFormat:@"V:|-100-[view(100)]" options:0 metrics:nil views:NSDictionaryOfVariableBindings(view)];
    [self.view addConstraints:constraintArray];
    [self.view addConstraints:constraintArray2];
}
```

上面代码中, 使用 `constraintsWithVisualFormat:options:metrics:views:` 方法用于根据 VFL 格式化字符串创建一系列的 `NSLayoutConstraint` 约束对象, 这些创建出来的约束对象会以数组的形式返回。要理解这个方法, 首先需要先明白 VFL 是什么。

VFL 即 `Visual-Format-Language` 格式化约束语言, 对应于上面代码的字符串。

```
H:|-20-[view(100@1000)]
V:|-100-[view(100)]
```

乍一看，VFL 确实十分令人费解，但是完全理解它之后就能感受到它的优美之处，它像极了中国古老的象形语言，通过半画半文字的方式表达信息。上面的第一条语句实际是约束了 view 视图左侧离父视图 20 个单位，宽度为 100 个单位，并且这条约束的优先级为 1000。第二条语句约束了 view 视图上侧离父视图 100 个单位，view 视图的高度为 100。现在可以解释一下 VFL 语言的语法意义了，最前面的 H 或者 V 代表约束的布局方向，H 是在水平方向添加约束，V 是在竖直方向添加约束。| 表示父视图的边缘，在 H 约束布局中，如果 | 出现在字符串的左端，则代表父视图的左边界，如果 | 出现在字符串的右端，则代表父视图的右边界；在 V 约束布局中，如果 | 出现在字符串的左边，代表父视图的上边界，如果 | 出现在字符串的右端，代表父视图的下边界。-x- 表示具体的约束距离，x 可为常量也可变为变量，常量例如前面示例中的 20，为变量的情况后面会有讨论。[] 内是布局摆放的控件名称，() 内是约束控件的尺寸，在 H 约束布局中，其意义是约束控件的宽度，在 V 约束布局中，其意义是约束控件的高度。@ 符号后面的值为设置此约束的优先级。

理解 VFL 语句的意义之后，再来看通过 VFL 语句创建约束集合的方法。

```
NSArray * constraintArray = [NSLayoutConstraint constraintsWithVisualFormat:@"H:|-20-[view(100@1000)]" options:0 metrics:nil views:NSDictionaryOfVariableBindings(view)];
```

constraintsWithVisualFormat:options:metrics:views:方法中第 1 个参数为创建约束的 VFL 字符串，第 2 个参数设置所约束控件的对齐模式，其需要设置为 NSLayoutConstraintOptions 类型的枚举，这个枚举中常用枚举值及意义如下所示。

```
typedef NS_OPTIONS(NSUInteger, NSLayoutConstraintOptions) {
    //约束的控件左对齐
    NSLayoutConstraintAlignAllLeft = (1 << NSLayoutConstraintAttributeLeft),
    //约束的控件右对齐
    NSLayoutConstraintAlignAllRight = (1 << NSLayoutConstraintAttributeRight),
    //约束的控件上对齐
    NSLayoutConstraintAlignAllTop = (1 << NSLayoutConstraintAttributeTop),
    //约束的控件下对齐
    NSLayoutConstraintAlignAllBottom = (1 << NSLayoutConstraintAttributeBottom),
    //约束的控件前对齐
    NSLayoutConstraintAlignAllLeading = (1 << NSLayoutConstraintAttributeLeading),
    //约束的控件后对齐
    NSLayoutConstraintAlignAllTrailing = (1 << NSLayoutConstraintAttributeTrailing),
    //约束的控件 Y 轴中心对齐
    NSLayoutConstraintAlignAllCenterX = (1 << NSLayoutConstraintAttributeCenterX),
    //约束的控件 X 轴中心对齐
    NSLayoutConstraintAlignAllCenterY = (1 << NSLayoutConstraintAttributeCenterY),
    //约束的控件文字基线对齐
    NSLayoutConstraintAlignAllBaseline = (1 << NSLayoutConstraintAttributeBaseline),
};
```


上面创建约束的方法中第4个参数为变量映射字典,如果VFL字符串中需要使用到某些变量,需要使用这个参数将变量映射到VFL字符串中,示例如下所示。

```
NSNumber * width = @20;
NSArray * constraintArray = [NSLayoutConstraint constraintsWithVisualForm
at:@"H:|-width-[view(100@1000)]" options:0 metrics:@{@"width":width} views:NS
DictionaryOfVariableBindings(view)];
```

上面示例代码中创建了一个NSNumber类型的变量,在metrics参数中,使用字典键值对的模式将VFL中对应的字符串应设置变量对象。

关于constraintsWithVisualFormat:options:metrics:views:方法中还有最后一个参数,views对应的参数是一个约束对象映射字典,与变量映射的原理一样,需要将VFL中使用到的具体控件名映射成视图控件对象。NSDictionaryOfVariableBindings()宏可以帮助开发者快捷的创建这个映射字典,开发者只需要将使用到的视图控件对象直接传入即可,有一点需要注意,使用NSDictionaryOfVariableBindings()宏进行快捷映射字典创建时,视图控件的名称必须和VFL中的名称完全一致。如果手写这个映射字典,上面的代码和下面所示的代码是等价的。

```
NSArray * constraintArray = [NSLayoutConstraint constraintsWithVisualForm
at:@"H:|-width-[view(100@1000)]" options:0 metrics:@{@"width":width} views:@
{@"view":view}];
```

8.2.5 与约束相关的几个方法

除了已经介绍的添加约束的方法外,autolayout框架中还提供了一些移除约束的方法,对于进行autolayout约束的视图控件而言,其中所有可用的有关设置约束的方法如下所示。

```
//为控件添加一个约束对象
- (void)addConstraint:(NSLayoutConstraint *)constraint ;
//为控件添加一组约束对象
- (void)addConstraints:(NSArray<__kindof NSLayoutConstraint *> *)constraints;
//移除控件上的一个约束对象
- (void)removeConstraint:(NSLayoutConstraint *)constraint;
//移除控件上的一组约束
- (void)removeConstraints:(NSArray<__kindof NSLayoutConstraint *> *)constraints;
```

8.2.6 使用 autolayout 设计一个高度自适应的聊天输入框及动画优化

许多与社交有关的应用都会有这样一个文本输入框,其默认高度是单行的,当用户输入的文字超出一行时,这个文本输入框会自适应成2行文本的高度,如果此时用户删除了一些输入的文字,使文字长度又集中在1行之内,则输入框的高度又会缩减为1行文本的高度,但是还有一点,如果用户输入的文字过多,行数超出了一定限度,文本输入框的高度会被固定变为内容可上下滚动的模式。使用autolayout约束布局并根据监听文本高度来更新布局可以十分轻松地完成一个这样的文本输入框。

使用 Xcode 创建一个名为 AutolayoutTestFive 的工程，使用 UITextView 来进行文本输入框的设计，现在 ViewController.m 文件中声明如下属性并遵守相应协议。

```
@interface ViewController ()<UITextViewDelegate>
{
    UITextView * _textView ;
    NSArray * _array1;
    NSArray * _array2;
}
@end
```

上面声明的属性中 _array1 与 _array2 用于存放约束对象。在 viewDidLoad 方法中添加如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    //进行文本输入框初始化
    _textView = [[UITextView alloc] init];
    _textView.layer.borderColor = [[UIColor grayColor] CGColor];
    _textView.layer.borderWidth = 1;
    _textView.translatesAutoresizingMaskIntoConstraints = NO;
    _textView.delegate=self;
    [self.view addSubview:_textView];
    //使用 VFL 进行约束的创建
    _array1 = [NSLayoutConstraint constraintsWithVisualFormat:@"H:|-100-
[_textView]-100-|" options:0 metrics:nil views:NSDictionaryOfVariableBindings
(_textView)];
    _array2 = [NSLayoutConstraint constraintsWithVisualFormat:@"V:|-150-
[_textView(30)]" options:0 metrics:nil views:NSDictionaryOfVariableBindings(_
textView)];
    //添加约束
    [self.view addConstraints:_array1];
    [self.view addConstraints:_array2];
}
```

在 ViewController.m 文件中实现 UITextViewDelegate 协议中的方法监听文本输入框中文字高度的变化，代码如下所示。

```
//监听 textView 中文本的变化
- (BOOL)textView:(UITextView*)textView shouldChangeTextInRange:(NSRange)range replacementText:(NSString *)text{
    //当文本高度大于 textView 的高度并且小于 100 时，更新约束
    if (textView.contentSize.height>textView.frame.size.height&&textView.contentSize.height<100) {
        float hight =textView.contentSize.height;
```

```

//将以前的移除掉
[self.view removeConstraints:_array1];
[self.view removeConstraints:_array2];
//创建新的约束
_array1 = [NSLayoutConstraint constraintsWithVisualFormat:@"H:|-100-[textView]-100-|" options:0 metrics:nil views:NSDictionaryOfVariableBindings(textView)];
_array2 = [NSLayoutConstraint constraintsWithVisualFormat:@"V:|-150-[textView](height)" options:0 metrics:@{@"height":[NSNumber numberWithIntFloat:height]} views:NSDictionaryOfVariableBindings(textView)];
[self.view addConstraints:_array1];
[self.view addConstraints:_array2];
}
//更新约束
[self.view updateConstraintsIfNeeded];
return YES;
}

```

上面代码中的 `updateConstraintsIfNeeded` 方法用于视图约束改变时调用来更新约束。

运行工程，可以看到文本框会随着文字的高度进行高度的自适应，但是仔细观察会发现，文本框的高度改变是在瞬间完成的，这给用户的感受将十分突兀，`autolayout` 实际上也是支持在 `UIView` 层动画中进行渐变动画的操作的，在上面方法调用 `updateConstraintsIfNeeded` 的地方后面添加如下代码，将会使 `autolayout` 约束的更新以过渡效果进行展示：

```

[UIView animateWithDuration:1 animations:^(
    [self.view layoutIfNeeded];
)];

```

8.2.7 使用第三方库 Masonry 进行 autolayout 约束布局

在前边章节的介绍中可以发现，无论是使用 `Objective-C` 风格的代码创建约束对象还是使用 `VFL` 字符串进行快捷结束都想创建都不能令开发者满意。`Objective-C` 风格的创建方法代码过于冗长，`VFL` 字符串的创建模式不便于编写与调试。在 `storyboard` 或者 `xib` 文件中简单拖拽的几条线使用原生代码实现却十分困难。在实际开发中，大多数开发者都会采用一个叫做 `Masonry` 的第三方库来协助进行代码约束的创建。

`Masonry` 是开源的第三方 `autolayout` 布局协助库，开发者可以使用 `cocoaPods` 或者在如下地址下载并将其集成在项目中。

```
https://github.com/SnapKit/Masonry
```

使用原生代码进行约束对象创建的时候，会使用如下这个核心方法。

```

+ (instancetype)constraintWithItem: (id)view1 attribute: (NSLayoutAttribute)attr1 relatedBy: (NSLayoutRelation)relation toItem: (nullable id)view2 attribute: (NSLayoutAttribute)attr2 multiplier: (CGFloat)multiplier constant: (CGFloat)c;

```


上面方法中 `NSLayoutAttribute` 类型的属性用于确定所约束的控件的某个属性, 在 Masonry 中, 有 `MASViewAttribute` 与其一一对应, 如图 8-24 表中所示。

MASViewAttribute	NSLayoutAttribute
view.mas_left	NSLayoutAttributeLeft
view.mas_right	NSLayoutAttributeRight
view.mas_top	NSLayoutAttributeTop
view.mas_bottom	NSLayoutAttributeBottom
view.mas_leading	NSLayoutAttributeLeading
view.mas_trailing	NSLayoutAttributeTrailing
view.mas_width	NSLayoutAttributeWidth
view.mas_height	NSLayoutAttributeHeight
view.mas_centerX	NSLayoutAttributeCenterX
view.mas_centerY	NSLayoutAttributeCenterY
view.mas_baseline	NSLayoutAttributeBaseline

图 8-24 `NSLayoutAttribute` 与 `MASViewAttribute` 属性对照表

Masonry 中为 `UIView` 类添加了一个类别, 这个类别中定义了 Masonry 中为管理视图 Autolayout 约束的 3 个核心方法。

下面方法用于为控件添加一组约束。

```
- (NSArray *)mas_makeConstraints:(void(^)(MASConstraintMaker *make))block;
```

上面方法为所调用它的视图控件创建一组 autolayout 约束, 在 block 代码块中进行具体的约束设置, 例如, 将控件的大小约束为 50×50, 位置约束在父视图的中心, 可以使用如下代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UILabel * label = [[UILabel alloc] init];
    [self.view addSubview:label];
    [label mas_makeConstraints:^(MASConstraintMaker *make) {
        //将视图控件的中心约束等于其父视图的中心
        make.center.equalTo(self.view);
        //约束控件的高度为 50 个单位
        make.height.equalTo(@50);
        //约束控件的宽度为 50 个单位
        make.width.equalTo(@50);
    }];
    label.backgroundColor = [UIColor redColor];
}
```

有一点需要注意, 使用 `mas_makeConstraints` 为视图控件创建约束时, 必须先将其添加到它的父视图上。可以发现, 创建同样的约束效果, Masonry 比系统原生的代码简洁许多。

如果需要更新一个视图控件的 `autolayout` 约束时, 可以使用如下方法。

```
[label mas_updateConstraints:^(MASConstraintMaker *make) {
    make.height.equalTo(@100);
    make.width.equalTo(@100);
}];
```

`mas_updateConstraints`: 方法用于更新一个视图控件所添加的约束, 开发者不需要将旧的约束移除, 直接将需要修改的约束条件写入 `block` 代码块中即可, 例如上面的代码将视图的宽度和高度约束修改为 100 个单位。

如果需要全部重新设置约束, 可以使用 `Masonry` 中的如下方法。

```
[label mas_remakeConstraints:^(MASConstraintMaker *make) {
    make.left.equalTo(self.view.mas_left).offset(10);
    make.top.equalTo(self.view.mas_top).offset(100);
    make.height.equalTo(@100);
    make.width.equalTo(@100);
}];
```

`mas_remakeConstraints`: 方法用于全部重设一个视图控件的 `autolayout` 约束。上面代码中 `offset()` 用于设置约束的偏移值, 例如上面代码的意义是将视图控件 `label` 约束在左侧离父视图左侧 10 个单位, 上侧离父视图上侧 100 个单位, 高度宽度约束为 100 个单位。

上面的示例代码中关于约束的关系, 使用的全部是精确的等于 `equalTo()`, 在 `Masonry` 中还有其他两种方式可以选择。

```
//大于等于
- (MASConstraintMaker * (^)(id attr))greaterThanOrEqualTo;
//小于等于
- (MASConstraintMaker * (^)(id attr))lessThanOrEqualTo;
```

在创建约束条件时也可以为这条约束设置优先值, 可使用的方法如下所示。

```
//手动设置一个优先级参数
- (MASConstraintMaker * (^)(MASLayoutPriority priority))priority;
//优先级低
- (MASConstraintMaker * (^)( ))priorityLow;
//优先级中等
- (MASConstraintMaker * (^)( ))priorityMedium;
//优先级高
- (MASConstraintMaker * (^)( ))priorityHigh;
```

示例代码如下所示。

```
[label mas_remakeConstraints:^(MASConstraintMaker *make) {
    make.left.equalTo(self.view.mas_left).offset(10);
    make.top.equalTo(self.view.mas_top).offset(100);
    //设置优先级为 1000
```

```

        make.height.equalTo(@100).priority(1000);
        //设置优先级高
        make.width.equalTo(@100).priorityHigh();
    }];

```

下面的示例为 Masonry 使用的几种案例。

1. 设置子视图与其父视图边距

使用 Xcode 创建一个名为 MasonryTestOne 的工程，在 ViewController.m 文件中引入 Masonry 库的头文件。

```
#import <Masonry/Masonry.h>
```



提示

如果 Masonry 是通过 cocoapods 进行引入的，由于 cocoapods 会使用静态库的方式管理第三方文件，因此在引入头文件时需要使用 `<>`，如果是开发者手动导入的 Masonry 库，则需要使用 `"`。

在 ViewController.m 文件的 viewDidLoad 方法中添加如下代码。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    UILabel * label = [[UILabel alloc] init];
    [self.view addSubview:label];
    [label mas_makeConstraints:^(MASConstraintMaker *make) {
        make.edges.equalTo(self.view).insets(UIEdgeInsetsMake(20, 20, 20, 20));
    }];
    label.backgroundColor = [UIColor redColor];
}

```

上面的代码将视图控件 label 约束上下左右边距父视图 20 个单位。运行工程，效果 8-25 所示。

2. 约束控件的尺寸为固定值

将所建工程中 viewDidLoad 中代码修改如下所示。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    UILabel * label = [[UILabel alloc] init];
    [self.view addSubview:label];
    [label mas_makeConstraints:^(MASConstraintMaker *make) {
        make.height.equalTo(@200);
        make.width.equalTo(@200);
        make.center.equalTo(self.view);
    }];
}

```



```
label.backgroundColor = [UIColor redColor];
```

上面代码将视图控件 label 尺寸约束为 200×200，位置约束在其父视图的中心，运行效果如图 8-26 所示。

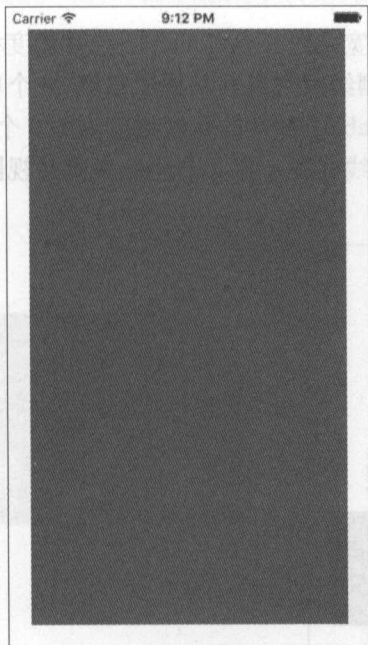


图 8-25 使用 Masonry 进行视图边距约束

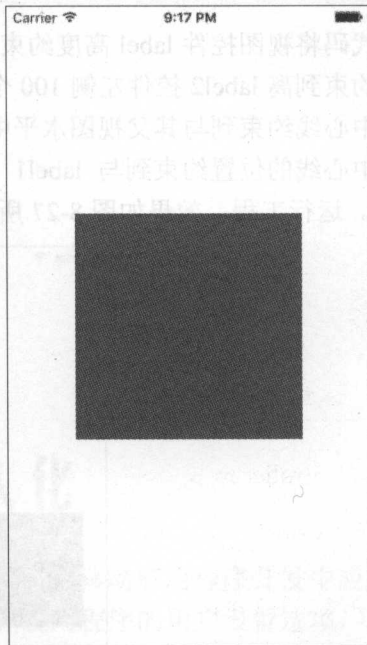


图 8-26 使用 Masonry 进行视图尺寸约束

3. 进行视图控件之间的约束

将所建工程中 viewDidLoad 方法的代码修改如下所示。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UILabel * label = [[UILabel alloc] init];
    [self.view addSubview:label];
    UILabel * label2 = [[UILabel alloc] init];
    [self.view addSubview:label2];
    [label mas_makeConstraints:^(MASConstraintMaker *make) {
        make.height.equalTo(@100);
        make.width.equalTo(label2);
        make.right.equalTo(label2.mas_left).offset(-100);
        make.leading.equalTo(self.view.mas_leading).offset(20);
        make.centerY.equalTo(self.view);
    }];
    [label2 mas_makeConstraints:^(MASConstraintMaker *make) {
        make.height.equalTo(@100);
        make.centerY.equalTo(label);
    }];
}
```

```
        make.trailing.equalTo(self.view.mas_trailing).offset(-20);  
    }  
    label.backgroundColor = [UIColor redColor];  
    label2.backgroundColor = [UIColor blueColor];  
}
```

上面代码将视图控件 label 高度约束为 100 个单位，宽度约束到与 label2 控件宽度相等，将其右侧约束到离 label2 控件左侧 100 个单位，将其左侧约束到离其父视图左侧 20 个单位，将其水平中心线约束到与其父视图水平中心线对齐。将 label2 控件的高度约束为 100 个单位，将其水平中心线的位置约束到与 label1 控件的水平中心线对齐，将其右侧约束离父视图右侧 20 个单位。运行工程，效果如图 8-27 所示。

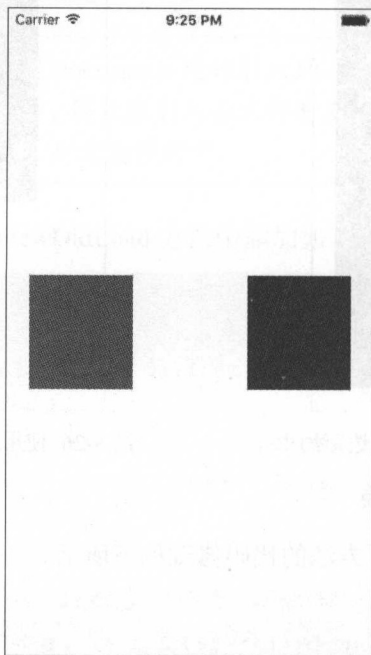


图 8-27 使用 Masonry 进行控件之间位置关系的约束

第 9 章

数据持久化

数据持久化处理在移动应用程序开发中应用十分广泛。例如用于保存应用程序的用户设置选项，保存一些非实时性网络请求的缓存数据，持久性数据的本地保存等。在 iOS 开发中，对于非常简单的小数据，可以使用 plist 文件或者数据归档技术进行持久化处理；如果需要本地化的数据结构非常复杂，数据量很大，通常可以采用本地数据库的方式来进行持久化；如果数据格式不定，灵活性较大，则可以采用写文件的方式来进行持久化。iOS 系统原生开发框架能够很好地支持小型数据库 SQLite，并且提供了面向对象的 CoreData 编程框架帮助开发者进行数据持久化的管理。

通过本章的学习，读者能够掌握：

1. 使用 plist 文件存储轻量级信息。
2. 使用归档的方式存储数据模型。
3. 能够进行简单的文件存取操作。
4. 使用 iOS 原生框架处理 SQLite 数据库。
5. 学会使用核心数据框架 CoreData。

9.1 使用 plist 文件进行轻量级数据持久化管理

对于 plist 文件，读者并不陌生。在前面的章节中，经常会提到一个工程配置文件 Info.plist。Info.plist 文件是开发者在创建工程的时候系统自动创建的一个配置文件，其中记录了许多工程设置选项的键值信息。plist 文件的结构十分简单，其内容的存储采用了 xml 数据的格式，因其后缀名为.plist，开发者也经常将其称为 plist 文件或者霹雳文件。

9.1.1 在工程中读取 plist 文件数据

使用 Xcode 创建一个名为 PlistTestOne 的工程，在 Xcode 的工程文件导航区单击右键创建一个新的文件，在弹出的新建文件窗口中选择 Resource 目录的 Property List 文件，将新创建的文件取名为 NewPlist，如图 9-1 所示。

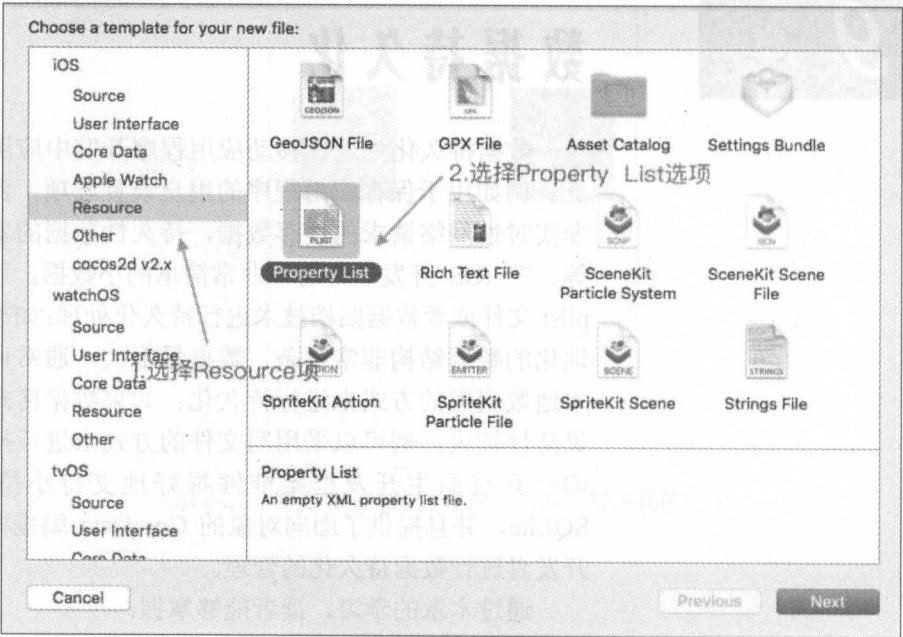


图 9-1 在工程中新建一个 plist 文件

单击创建的 NewPlist 文件，会看到如图 9-2 所示的编辑界面。这个 plist 文件有一个默认根节点，开发者可以自行将其设置为 Array 类型或者 Dictionary 类型。

Key	Type	Value
Root	Dictionary	(0 items)

图 9-2 plist 文件默认根节点

向 NewPlist 文件中添加一组测试数据，如图 9-3 所示。

Key	Type	Value
▼ Root	Dictionary	{3 Items}
isSuccess	Boolean	YES
count	Number	100
Name	String	测试

图 9-3 向 NewPlist 文件中添加一组数据

图 9-3 中向 NewPlist 文件的根节点中添加了 3 个子节点, 1 个是 Boolean 类型的 isSuccess 字段, 1 个是 NSNumber 类型的 count 字段, 1 个是 String 类型的 Name 字段。下面, 将使用代码在程序中获取 NewPlist 文件中的数据。

在工程 ViewController.m 文件的 viewDidLoad 方法中添加如下测试代码。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    //获取 plist 文件路径
    NSString * path = [[NSBundle mainBundle]pathForResource:@"NewPlist" of
Type:@"plist"];
    //获取 plist 文件内容
    NSDictionary * rootDic = [NSDictionary dictionaryWithContentsOfFile:pa
th];
    //遍历打印
    for (NSString * key in rootDic.allKeys) {
        NSLog(@"%@:%@", key, rootDic[key]);
    }
}

```

运行工程, 可以在 Xcode 的调试区看到如图 9-4 所示的信息, 说明已经将 NewPlist.plist 文件中的数据读取到了程序中。

```

2016-03-30 19:05:32.592 PlistTestOne[1000:50304] count:100
2016-03-30 19:05:32.593 PlistTestOne[1000:50304] Name:测试
2016-03-30 19:05:32.593 PlistTestOne[1000:50304] isSuccess:1

```

图 9-4 Xcode 的打印信息



提示

在工程中直接创建的 plist 文件会被打包进 Main Bundle 中, 开发者只能在程序中读取, 但没有权限对其进行写操作。如果要在程序中使用可读可写的 plist 文件, 需要在应用沙盒的 Documents 文件夹中进行, 这部分内容将在下一小节进行介绍。

9.1.2 在程序沙盒 Documents 目录中创建和使用 plist 文件

所谓沙盒是 iOS 保护用户信息安全的一种体制, 它限定应用程序只能在特定的目录内读写数据, 不可访问其他地方的内容。每个应用程序都有自己的文件沙盒, 下面将在当前应用程序沙盒的 Documents 文件夹中创建 plist 文件进行存取操作。Documents 文件夹常用于存放应用程序中文档类型相关的数据。

使用 Xcode 创建一个名为 PlistTestTwo 的工程，在 ViewController.m 文件的 viewDidLoad 方法中添加如下测试代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    //获取沙盒目录
    NSArray *paths=NSSearchPathForDirectoriesInDomains(NSDocumentDirector
y,NSUserDomainMask,YES);
    NSString *plistPath1 = [paths objectAtIndex:0];
    //拼接得到完整的文件名
    NSString *filename=[plistPath1 stringByAppendingPathComponent:@"my.pl
ist"];
    NSDictionary * dic = @{@"my":@"test"};
    //将数据写入文件
    [dic writeToFile:filename atomically:YES];
    //取数据
    NSDictionary * getDic = [NSDictionary dictionaryWithContentsOfFile:fil
ename];
    NSLog(@"%@",getDic);
}
```

上面代码将字典写入一个沙盒 Documents 文件夹中的一个 plist 文件，在将数据从中取出，运行工程后，Xcode 的调试区打印信息如图 9-5 所示。

```
2016-03-30 19:46:58.703 PlistTestTwo[1201:67346]
{
    my = test;
}
```

图 9-5 Xcode 调试区打印的内容

9.1.3 使用 NSUserDefaults 类进行数据持久化

NSUserDefaults 是 iOS 开发框架中提供给开发者使用的面向对象的数据持久化接口，通过 NSUserDefaults 类，开发者可以十分方便地进行数据存取管理。实际上，NSUserDefaults 内部也是采用 plist 文件进行数据的存储操作的。

NSUserDefaults 采用单例的设计模式，整个应用程序共享一个 NSUserDefaults 对象。NSUserDefaults 通常用于存储应用程序中相关用户设置的数据，也可用其来进行应用中数据的本地保存与传递。

可以用如下代码来获取系统的 NSUserDefaults 单例对象。

```
NSUserDefaults * defaults = [NSUserDefaults standardUserDefaults];
```

NSUserDefaults 采用键值的方式来进行数据的存取对应，关于 NSUserDefaults 的存取数据操作，系统提供了一系列方法分别用于存取相应类型的数据，代码如下所示。

```
//获取字符串数据
- (NSString *)stringForKey:(NSString *)defaultName;
```



```

//获取数组数据
- (NSArray *)arrayForKey:(NSString *)defaultName;
//获取字典数据
- (NSDictionary *)dictionaryForKey:(NSString *)defaultName;
//获取 NSData 类型数据
- (NSData *)dataForKey:(NSString *)defaultName;
//获取字符串数组数据
- (NSArray *)stringArrayForKey:(NSString *)defaultName;
//获取整型数据
- (NSInteger)integerForKey:(NSString *)defaultName;
//获取浮点型数据
- (float)floatForKey:(NSString *)defaultName;
//获取双精度浮点型数据
- (double)doubleForKey:(NSString *)defaultName;
//获取布尔类型数据
- (BOOL)boolForKey:(NSString *)defaultName;
//获取 URL 数据
- (NSURL *)URLForKey:(NSString *)defaultName;
//获取对象数据
- (id)objectForKey:(NSString *)defaultName;

```

相对于上面用来获取数据的 `getter` 方法，其对应的存储数据的 `setter` 方法如下所示。

```

//存储对象数据
- (void)setObject:(id)value forKey:(NSString *)defaultName;
//存储整型数据
- (void)setInteger:(NSInteger)value forKey:(NSString *)defaultName;
//存储浮点型数据
- (void)setFloat:(float)value forKey:(NSString *)defaultName;
//存储双精度浮点型数据
- (void)setDouble:(double)value forKey:(NSString *)defaultName;
//存储布尔类型数据
- (void)setBool:(BOOL)value forKey:(NSString *)defaultName;
//存储 URL 类型数据
- (void)setURL:(NSURL *)url forKey:(NSString *)defaultName;

```

使用 Xcode 创建一个名为 `NSUserDefaultsTest` 的工程，在 `ViewController.m` 文件的 `viewDidLoad` 方法中编写如下测试代码。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    //获取 NSUserDefaults 单例对象
    NSUserDefaults * defaults = [NSUserDefaults standardUserDefaults];
    //进行数据存储
    [defaults setObject:@"测试数据" forKey:@"test"];
    //同步到磁盘
}

```

```

//获取数据
NSString * str = [defaults objectForKey:@"test"];
NSLog(@"%@",str);
}

```

上面代码中，NSUserDefaults 对象调用 `synchronize` 方法将存储的数据同步保存到本地磁盘。为了保证数据的顺利存储，在 NSUserDefaults 对象调用 `setter` 方法进行数据存储后，开发者尽量都手动调用一次 `synchronize` 方法。此时运行工程，根据打印信息可以判断数据存取是否成功，实际上，数据一旦被存入 NSUserDefaults，就算应用程序被退出。再次开启的应用程序 NSUserDefaults 中存储的数据不会丢失，运行一次程序后，将 `viewDidLoad` 方法中的代码修改如下，运行工程，可以看到依然从 NSUserDefaults 中读取到了数据。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    //获取 NSUserDefaults 单例对象
    NSUserDefaults * defaults = [NSUserDefaults standardUserDefaults];
    //获取数据
    NSString * str = [defaults objectForKey:@"test"];
    NSLog(@"%@",str);
}

```

9.2 使用归档技术进行数据模型持久化

在上一小节中，介绍了使用 `plist` 文件进行数据持久化的方法，虽然十分简单方便，但只能支持基本数据类型和 Objective-C 中简单对象的存取操作，对于一些复杂类对象与开发者自定义的数据模型并不能使用 `plist` 文件进行存储。然而在实际开发中，开发者经常需要对一些自定义的类对象与数据模型进行持久化存储，例如在游戏中角色的属性、装备等模式数据的存储，网络缓存数据的存储等。

iOS 编程中的归档技术实际上并非对数据进行持久化处理，归档是一种编码与序列化方式，它可以将复杂的数据模型转换为二进制数据，通过结合文件读写操作来进行二进制数据的存储。当读取数据时，也是先将文件中的二进制数据读取出来，然后运用解归档技术对二进制数据进行反序列化来解归档出原始的数据模型。有了归档技术，开发者可以更加灵活地对复杂数据进行持久化操作。

9.2.1 进行单一系统数据类型的归档与解归档操作

类似于 NSUserDefaults 类中的单例对象，iOS 归档工具类也提供了一个根键值用于开发者进行快捷的数据归档操作。

使用 Xcode 创建一个名为 `ArchiverTestOne` 的工程，在 `ViewController.m` 文件的 `viewDidLoad` 方法编写如下测试代码。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    //获取根目录
    NSString *homeDictionary = NSHomeDirectory();
    //为根目录拼接上储存的文件名
    NSString *homePath = [homeDictionary stringByAppendingPathComponent:@"atany.archiver"];
    //归档方式一：通过 data 数据归档，在将数据写入文件
    NSData *data= [NSKeyedArchiver archivedDataWithRootObject:@"123"];
    [data writeToFile:homePath atomically:YES];
    //方式二：直接写入文件
    [NSKeyedArchiver archiveRootObject:@"456" toFile:homePath];
    //方式一和方式二的效果完全一样 只是解归档的时候不同
    //方式一的解归档：先获取 data 数据，在进行 data 数据的解归档
    NSLog(@"%@", [NSKeyedUnarchiver unarchiveObjectWithData:data]);
    //方式二的解归档：直接解文件中的归档
    NSLog(@"%@", [NSKeyedUnarchiver unarchiveObjectWithFile:homePath]);
}

```

上面代码中采用两种方式进行归档与解归档的演示，前面提到，归档只是对数据的一种序列化方式，将归档的数据写入文件才能保证数据的持久化存储。NSKeyedArchiver 类用于数据进行归档，其 archivedDataWithRootObject:方法可以将 Objective-C 基本的数据对象序列化为 NSData 数据，NSData 类型的数据对象可以调用 writeToFile:方法来将数据写入磁盘文件。NSKeyedArchiver 类的 archiveRootObject:toFile:方法则是直接执行了两步操作，首先将数据序列化为 NSData 数据，在将数据直接写入参数提供的磁盘路径文件中。与 NSKeyedArchiver 类相对应，NSKeyedUnarchiver 类用于将归档后的数据还原为原数据类型，这一过程也被称为解归档。NSKeyedUnarchiver 类的 unarchiveObjectWithData:方法用于将归档后的 NSData 数据还原为数据。NSKeyedUnarchiver 类的 unarchiveObjectWithFile:方法则是直接将路径文件的数据读出并实现解归档操作。

上面演示的方法都是对单一数据进行归档操作，然而在实际应用中，这种情况十分少见。大多数情况下，开发者都需要对多种数据类型混合的数据进行统一归档操作，例如，归档一个学生对象就需要归档这个学生字符串数据类型名字和整型数据类型的年龄等。下一小节将介绍如何在同一归档数据中归档存储多个数据。

9.2.2 对多个对象进行数据归档

对多个数据进行统一的数据归档，需要开发者自己构造归档类，解归档时亦然。

使用 Xcode 创建一个名为 ArchiverTestTwo 的工程，在 ViewController.m 文件的 viewDidLoad 方法中添加如下测试代码。

```

- (void)viewDidLoad {
    [super viewDidLoad];

```



```

//获取根目录
NSString *homeDictionary = NSHomeDirectory();
//添加储存的文件名
NSString *homePath = [homeDictionary stringByAppendingPathComponent:@"atany.archiver"];
//这里创建一个可变的 data 对象作为归档的容器
NSMutableData * data = [[NSMutableData alloc] init];
//创建一个归档对象，归档后写入 data 数据
NSKeyedArchiver * archiver = [[NSKeyedArchiver alloc] initWithWritingWithMutableData:data];
//对下面的字符串和 int 值进行归档序列化
[archiver encodeObject:@"jaki" forKey:@"name"];
[archiver encodeInt:24 forKey:@"age"];
//写入 data
[archiver finishEncoding];
//写入文件
[data writeToFile:homePath atomically:YES];
//创建解归档的反序列化对象
NSKeyedUnarchiver * unarchiver = [[NSKeyedUnarchiver alloc] initWithData:data];
//进行反序列化
NSString * name = [unarchiver decodeObjectForKey:@"name"];
int age = [unarchiver decodeIntForKey:@"age"];
//打印信息
NSLog(@"\nname:%@\nage:%d", name, age);
}

```

通过上面的方法，可以实现对不同数据类型的多个数据的统一归档操作，其实任何复杂的数据对象都是由基础的数据对象所组成的，有了这样的思路，使实现自定义对象的归档操作成为可能。

9.2.3 进行自定义数据模型的归档

在 9.2.2 小节中，介绍了对多数据类型统一归档的方法，然而依然有很大的缺陷。开发者编写程序时，有时一个自定义的数据对象会十分复杂庞大，如果使用 9.2.3 中的方法，没对这个数据对象进行一次解归档操作会将十分麻烦。实际上，在 iOS 开发框架中有一个名叫 NSCoding 的协议，任何遵守 NSCoding 协议并且实现了协议方法的类都可以支持归档与解归档操作。

使用 Xcode 创建一个名为 ArchiveTestThree 的工程，在工程中新建一个类，命名为 Model，使其继承与 NSObject。在 Model.h 文件中编写如下代码。

```

@interface Model : NSObject<NSCoding>
@property(nonaatomic,strong)NSString * name;
@property(nonaatomic,assign)int age;
@end

```

在 Model.m 文件中实现 NSCoder 协议中的方法如下所示。

```
@implementation Model
//解档方法
- (instancetype)initWithCoder:(NSCoder *)coder
{
    if (self=[super init]) {
        _name = [coder decodeObjectForKey:@"name"];
        _age = [coder decodeIntForKey:@"age"];
    }
    return self;
}
//归档方法
- (void)encodeWithCoder:(NSCoder *)coder
{
    [coder encodeObject:_name forKey:@"name"];
    [coder encodeInt:_age forKey:@"age"];
}
@end
```

在上面的代码中，initWithCoder:方法当对象被解归档时会被调用，在其中需要将对象的属性分别解归档出来。encodeWithCoder:方法当对象被归档时会被调用，在其中需要将对象的属性分别都进行归档操作。

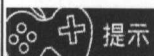
在 ViewController.m 文件引入如下头文件。

```
#import "Model.h"
```

在 ViewController.m 文件的 viewDidLoad 方法中添加如下测试代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    Model * obj = [[Model alloc] init];
    obj.name = @"jaki";
    obj.age = 24;
    //进行自定义对象的归档
    NSData * data = [NSKeyedArchiver archivedDataWithRootObject:obj];
    //进行解档
    Model * obj2 = [NSKeyedUnarchiver unarchiveObjectWithData:data];
    NSLog(@"\name:%@\nage:%d", obj2.name, obj2.age);
}
```

运行工程，可以根据打印信息看出对自定义对象的归档是否成功。



如果自定义的数据模型类并没有实现 NSCoder 中的归档与解归档方法，程序是会崩溃的。

9.3 小型数据库 SQLite 在 iOS 开发中的应用

数据库是一种数据管理工具,通过特有的语法语言与命令可以对数据库中的数据进行管理操作。在移动开发中,通常会用到一些小型的数据库进行数据管理。SQLite 是一款十分小巧便捷的数据库,在 iOS 开发中,原生开发框架也对其有很好的支持。

9.3.1 SQLite 数据库常用语法介绍

SQLite 数据库的操作是由 SQL 语句完成的,开发者若要在移动应用中嵌入 SQLite 数据库进行使用,了解一些基本的 SQL 语法是必不可少的。

数据库是由一张一张具体的表组成的,在 SQLite 数据库中,可用如下语句来建立一张表。

```
create table class(num integer PRIMARY KEY,name text NOT NULL DEFAULT "1班",count integer CHECK(count>10))
```

上面的语句可以简化为下面的格式。

```
creat table tableName(param1 type key,param2 type key,...)
```

上面语句格式中,creat table 是固定语句,用于在数据库中创建一张数据表,之后的参数需要开发者自行设置,tableName 为所创建的表的表名,小括号内为表中提供的数据字段,多个字段以逗号进行分割,其中 param 是字段名称,type 是字段对应的数据类型,key 是字段的修饰符.SQLite 数据库中支持的数据类型有如下几种。

- smallint 短整型
- integer 整型
- real 实数型
- float 单精度浮点
- double 双精度浮点
- currency 长整型
- varchar 字符型
- text 字符串
- binary 二进制数据
- blob 二进制大对象
- boolean 布尔类型
- date 日期类型
- time 时间类型
- timestamp 时间戳类型

关于数据的修饰符,SQLite 数据库中常用如下所示。

- PRIMARY KEY: 将本参数设置为主键,主键的值必须唯一,可以作为数据的索引,例如编号。

- NOT NULL : 标记本参数为非空属性。
- UNIQUE: 标记本参数的键值唯一, 类似主键。
- DEFAULT: 设置本参数的默认值
- CHECK: 参数检查条件, 例如上面代码, 写入数据时 count 必须大于 10 才有效。

使用如下语句向 SQLite 数据库表中添加一条数据。

```
insert into class(num,name,count) values(2,"三年 2 班",58)
```

上面语句可以简化为下面的格式:

```
insert into tableName(param1,param2,...) values(value1,value2,...)
```

上面语句格式中, insert into 为插入语句的固定格式, tableName 需要填写为要插入数据的表名, 第 1 个小括号内对应插入此条数据的各个参数名, values 后面小括号中需要填写各个参数名所对应的值。

如果要想 SQLite 表中添加一个新的字段, 使用如下语句:

```
alter table class add new text
```

上面语句可以简化为如下格式:

```
alter table tableName add keyName type
```

上面语句格式中, alter table 为固定格式, tableName 为要添加字段的表名, add 为固定格式, keyName 为要添加的字段名称, type 为要添加的字段类型。

在 SQLite 数据库中, 使用如下的语句进行数据修改。

```
update class set num=3,name="新的班级" where num=1
```

上面语句可以简化为如下格式:

```
update tableName set param1=value1, param2=value2 where require
```

上面语句格式中, update 为修改数据的固定格式, tableName 为要修改的数据所在的表名, set 后面需要填写所修改字段的键值对, where 为固定格式, require 为约束条件, 例如上面的示例中将 num 值为 1 的数据进行修改操作。

在 SQLite 中使用如下语句进行删除数据操作。

```
delete from class where num=1
```

上面语句可以简化为如下格式:

```
delete from tableName where require
```

上面语句格式中, delete from 为删除数据的固定格式, tableName 为要删除数据的表名, where 为固定格式, require 为约束条件, 例如上面的示例中将 num=1 的数据删除。

上面介绍了操作数据库的增、删、改操作, 其实数据库中还有一个重要的功能也是最核心的功能“查”, SQLite 中提供了许多命令语句来完成复杂的数据查询功能。

查询数据库表中的某个字段数据的值。

```
select num from class
```

上面语句可以简化为如下格式：

```
select key from tableName
```

上面语句格式中，`select` 为查询语句的固定格式，`key` 为要查询的字段，`from` 为固定格式，`tableName` 为要查询的表名。

如果要查询表中所有字段的数据，使用如下格式的语句：

```
select* from tableName
```

符号*在 SQLite 数据库中充当着通配符的角色，用其来代替一个或多个真正字符。SQLite 也支持对查询的数据进行排序操作，示例如下所示。

```
select * from class order by count asc
```

上面语句可以简化为如下格式：

```
select key from tableName order by key param
```

带排序的查询语句后面多了一些参数，其中 `order by` 为排序语句的固定格式，`key` 为要进行排序的字段，`param` 设置排序类型，如果设置为 `asc` 则为升序排序；如果设置为 `desc` 则为降序排序。

有时一个数据库文件中存储的数据量是十分巨大的，一张表中的数据条数也可能成千上万，在移动应用开发中，开发者通常并不需要一次将所有数据都查询出来，SQLite 也提供了设置数据查询位置与条数的方法，示例语句如下所示。

```
select * from class limit 2 offset 0
```

上面语句可以简化为如下格式：

```
select key from tableName limit paramlit offset paramset
```

`limit` 与 `offset` 为固定格式，`paramlit` 为设置查询数据的条数限制，`paramset` 设置从第几条数据开始进行查询。

9.3.2 使用 iOS 原生框架 sqlite3 对 SQLite 数据库进行操作

iOS 的原生开发框架可以对 SQLite 数据库进行很好的支持，sqlite3 框架中采用 C 函数的设计风格而且通过指针移动来进行 SQLite 数据库的相关操作。

使用 Xcode 创建一个名为 SQLiteTest 的工程，libsqlite3 是对 SQLite 数据库进行操作的系统库，在使用前，需要先将这个框架导入工程。单击 Xcode 配置文件中的 Build Phases 标签，将其中的 Link Binary With Libraries 展开，单击其中的加号进行系统框架的引入，过程如图 9-6 所示。

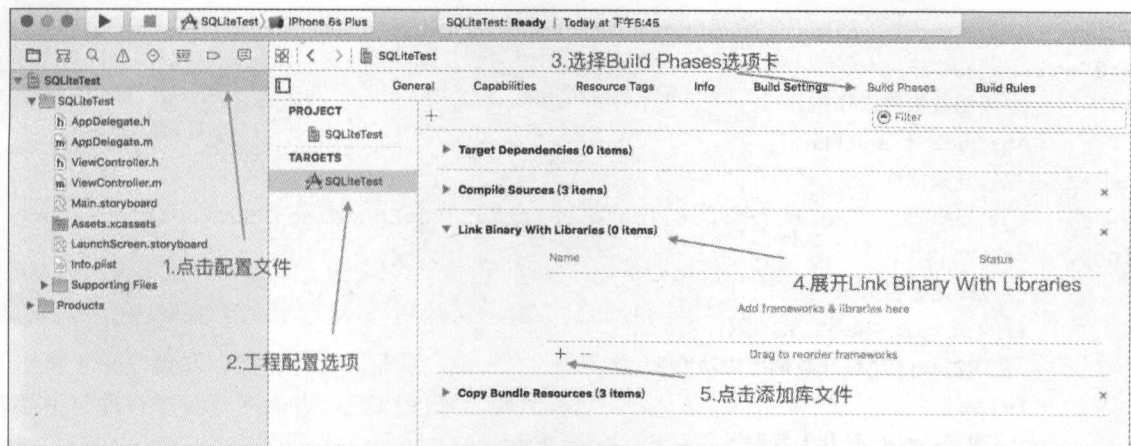


图 9-6 在工程导入库文件过程

在弹出的窗口中找到并选中 libsqlite3.0 库后，单击 add 按钮进行添加，如图 9-7 所示。

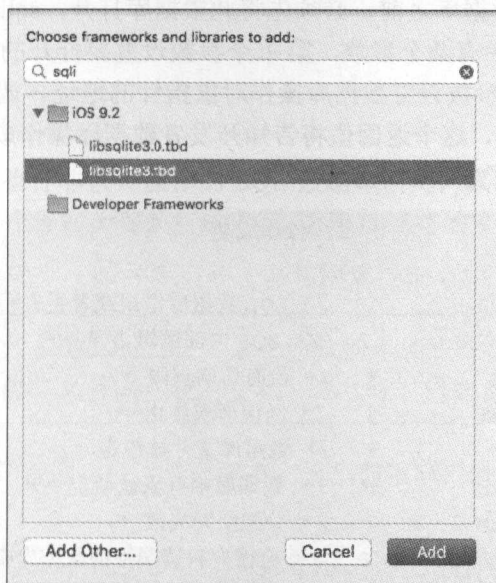


图 9-7 将 libsqlite3 库文件引入工程中

在工程 ViewController.m 文件中引入如下头文件来使用 libsqlite3 系统库。

```
#import <sqlite3.h>
```

要对 SQLite 数据库进行学习演示，首先需要向工程中引入一个 SQLite 数据库文件，这里演示的数据库文件为驾照科目二考试的题库数据库，读者可以在本书配套的代码素材中找到此数据库文件。

在 ViewController.m 文件的 viewDidLoad 方法中添加如下演示代码。

```
-(void)viewDidLoad {
    [super viewDidLoad];
    // 获取数据库路径
```



```

NSString * path = [[NSBundle mainBundle] pathForResource:@"data" ofType:@"sqlite"];
//声明数据库操作指针
sqlite3 * sqlite;
//打开数据库
int result = sqlite3_open([path cStringUsingEncoding:NSUTF8StringEncoding], &sqlite);
//判断是否打开成功
if (result==SQLITE_OK) {
    NSLog(@"打开数据库成功");
}else{
    NSLog(@"打开失败");
}
}

```

在操作某个 SQLite 数据库之前，必须先将此数据库打开，sqlite3_open()方法用于打开一个数据库文件，这个方法中有两个参数，第 1 个参数设置要打开的数据库地址，必须为 C 语言类型的字符串，第 2 个参数设置数据库操作对象指针的地址。调用 sqlite3_open()方法后会返回一个 int 类型的返回值，这个返回值将告知开发者数据库操作成功或者不成功的原因，int 值所对应的意义都定义成了宏，如下所示。

```

#define SQLITE_OK 0 //操作成功
/* 以下是错误代码 */
#define SQLITE_ERROR 1 /* SQL 数据库错误或者丢失 */
#define SQLITE_INTERNAL 2 /* SQL 内部逻辑错误 */
#define SQLITE_PERM 3 /* 没有访问权限 */
#define SQLITE_ABORT 4 /* 回调请求终止 */
#define SQLITE_BUSY 5 /* 数据库文件被锁定 */
#define SQLITE_LOCKED 6 /* 数据库中有表被锁定 */
#define SQLITE_NOMEM 7 /* 分配空间失败 */
#define SQLITE_READONLY 8 /* 企图向只读属性的数据库上做写操作 */
#define SQLITE_INTERRUPT 9 /* 通过 sqlite3_interrupt() 方法终止操作 */
#define SQLITE_IOERR 10 /* 磁盘发生错误 */
#define SQLITE_CORRUPT 11 /* 数据库磁盘格式不正确 */
#define SQLITE_NOTFOUND 12 /* 调用位置操作码 */
#define SQLITE_FULL 13 /* 由于数据库已满造成的添加数据失败 */
#define SQLITE_CANTOPEN 14 /* 不法打开数据库文件 */
#define SQLITE_PROTOCOL 15 /* 数据库锁协议错误 */
#define SQLITE_EMPTY 16 /* 数据库为空 */
#define SQLITE_SCHEMA 17 /* 数据库模式更改 */
#define SQLITE_TOOBIG 18 /* 字符或者二进制数据超出长度 */
#define SQLITE_CONSTRAINT 19 /* 违反协议终止 */
#define SQLITE_MISMATCH 20 /* 数据类型不匹配 */
#define SQLITE_MISUSE 21 /* 库使用不当 */

```

```

#define SQLITE_NOLFS      22  /* 使用不支持的操作系统 */
#define SQLITE_AUTH      23  /* 授权拒绝 */
#define SQLITE_FORMAT    24  /* 辅助数据库格式错误 */
#define SQLITE_RANGE     25  /* sqlite3_bind 第二个参数超出范围 */
#define SQLITE_NOTADB    26  /* 打开不是数据库的文件 */
#define SQLITE_NOTICE    27  /* 来自 sqlite3_log() 的通知 */
#define SQLITE_WARNING   28  /* 来自 sqlite3_log() 的警告 */
#define SQLITE_ROW       100 /* sqlite3_step() 方法准备好了一行数据 */
#define SQLITE_DONE      101 /* sqlite3_step() 已完成执行 */

```

对于一个数据库的操作，无非增、删、改、查4种，在打开一个有编辑权限的数据库之后，若要执行非查询类型的操作，使用如下示例代码。

```

//创建错误信息指针
char * error;
NSString * sqlStr = @"create table Class(num integer,name text)";
//执行 SQL 语句
result = sqlite3_exec(sqlite, [sqlStr cStringUsingEncoding:NSUTF8String
                          Encoding], NULL, NULL, &error);
NSLog(@"%d:%s",result,error);

```

注意 `Sqlite3_exec()`方法用于执行非查询操作的 SQL 数据库语句，其中第1个参数为已成功打开数据库的操作指针，第2个参数为要执行的 SQL 语句 C 风格的字符串，最后一个参数为错误信息指针的地址。



提示

只有在应用沙盒 Documents 目录中的数据库文件才有被修改的权限，直接导入工程的数据库文件只能进行查找，不可进行修改操作。

使用 `libsqlite3` 框架进行查询操作，略显复杂，这里需要额外的一个指针记录数据查询的位置，示例代码如下所示。

```

sqlite3_stmt * stmt = nil;
NSString * searchSql = @"select * from firstlevel";
result = sqlite3_prepare(sqlite, [searchSql cStringUsingEncoding:NSUTF8String
                               Encoding], -1, &stmt, NULL);
if (result==SQLITE_OK) {
    while (sqlite3_step(stmt)==SQLITE_ROW) {
        char * pname = (char *)sqlite3_column_text(stmt, 2);
        NSLog(@"%@", [NSString stringWithCString:pname encoding:NSUTF8String
                                                    Encoding]);
    }
    sqlite3_finalize(stmt);
}

```

首先，工程中导入的数据库文件中有 firstlevel 这样一张表，这张表中的数据结构如图 9-8 所示。

serial	pid	pname	pcount
1	1	道路交通安全法律、法规和规章(185题)	16
2	2	交通信号及其含义(158题)	8
3	3	安全行车、文明驾驶知识(159题)	8
4	4	高速公路、山区道路、桥梁、隧道、夜	10
5	5	出现爆胎、转向失控、制动失灵等紧急	23
6	6	机动车总体构造和主要安全装置常识、	6
7	7	发生交通事故后的自救、急救等基本知	4

图 9-8 firstlevel 表中的数据结构

上面的代码中，stmt 是一个标记数据位置的指针 sqlite3_prepare()方法用于做数据查询前的准备，其中第 1 个参数为成功打开的数据库操作指针，第 2 个参数为数据库查询 SQL 语句 C 风格的字符串，第 3 个参数为数据库位置 stmt 指针，这个方法的返回值告知开发者准备查询工作是否执行成功。sqlite3_step()方法用于将 stmt 指针进行逐行移动，这个方法返回的值如果是 SQLITE_ROW 所定义的值，则表明此行有数据，开发者可以通过这个指针进行数据库数据的读取。sqlite3_column_text()方法用于取数据库中 text 类型字段的值，其中第 1 个参数为 stmt 指针，第 2 个参数为所取字段在数据表中的序号，例如上面数据表中 text 类型的 pname 字段为数据库表中的第 3 个字段，这里的参数从 0 开始，因此应该设置为 2。sqlite3_column_text()方法还有一系列的姊妹方法，这些方法用于取数据库中相应类型字段的值，常用方法如下所示。

```
//取布尔类似字段的值
SQLITE_API const void *SQLITE_STDCALL sqlite3_column_blob(sqlite3_stmt*,
int iCol);

//取 Byte 类型字段的值
SQLITE_API int SQLITE_STDCALL sqlite3_column_bytes(sqlite3_stmt*, int iCol);
//取 Byte16 类型字段的值
SQLITE_API int SQLITE_STDCALL sqlite3_column_bytes16(sqlite3_stmt*, int iCol);
//取 double 类型字段的值
SQLITE_API double SQLITE_STDCALL sqlite3_column_double(sqlite3_stmt*, int
iCol);
//取 int 类型字段的值
SQLITE_API int SQLITE_STDCALL sqlite3_column_int(sqlite3_stmt*, int iCol);
//取 int64 类型字段的值
SQLITE_API sqlite3_int64 SQLITE_STDCALL sqlite3_column_int64(sqlite3_stmt
*, int iCol);
//取 text 类型字段的值
SQLITE_API const unsigned char *SQLITE_STDCALL sqlite3_column_text(sqlite
3_stmt*, int iCol);
//取 text16 类型字段的值
SQLITE_API const void *SQLITE_STDCALL sqlite3_column_text16(sqlite3_stmt*,
int iCol);
```




提示

1. 使用完 stmt 指针后, 要调用 `sqlite3_finalize()` 方法对 stmt 指针进行关闭。
2. `NSStringEncoding` 是对字符串进行编码的一种方式, 其可以支持中文字符的显示。

9.4 核心数据管理框架 CoreData 的使用

CoreData 是 iOS 开发框架中一个专门用来提供和管理数据服务的编程框架, 无论在性能上还是书写方便上, 都是开发者的首选。在数据管理方面, Apple 强烈推荐开发者使用 CoreData 框架, 并且官方文档中称正确熟练地使用 CoreData 框架可以为开发者简化 50% 以上的代码量。虽然 iOS 开发框架中的 `libsqlite3` 框架可以很好地支持 SQLite 数据库, 但是其 C 语言风格的函数和编程思路穿插在 Objective-C 代码中既显得生硬, 编写也复杂, CoreData 则不同, 它采用面向对象的设计风格, 加上与 Xcode 的可视化数据操作工具结合使用, 极大地方便了开发者在应用开发中设计模型与处理数据。

9.4.1 使用 CoreData 设计数据模型

在 iOS 开发中, 对于大数据量数据的处理, SQLite 是一个不错的选择, 其对独立数据类型的支持没有缺陷, 例如一张学生表中可以存放学生的学号、姓名、年龄等信息, 一张班级表中可以存放班级的名称、人数等信息。但是如果学生表和班级表产生了联系, 例如一张班级表中不仅有班级人数、名称信息, 往往还有这个班级中每个学生的信息, 要在学生表和班级表直接建立一定的联系, 使用 SQLite 就会复杂很多, 抽象一点来说, 班级表与学生表之间产生联系的实质就是数据模型与数据模型之间的嵌套, 处理这种类型的数据, 正是 CoreData 框架的优势所在。

使用 Xcode 开发工具创建一个名为 CoreDataTest 的工程, 在 Xcode 文件导航区单击右键, 创建一个新的文件。在弹出的文件创建窗口中选中 Core Data 选项, 选择其中的 Data Model, 即数据模型文件进行创建, 这里取名为 StudentModel, 创建步骤如图 9-9 所示。

此时在 Xcode 的文件导航区可以看到多了一个 StudentModel.xcdatamodeld 文件, 这个文件就是上面创建的学生数据模型文件。单击这个文件, 可以看到 Xcode 的编码区变为数据模型工具界面, 单击 Add Entity 按钮为 StudentModel 数据模型文件添加一个实体, 如果和 SQLite 数据库向类比, CoreData 数据模型文件的实体就如 SQLite 数据库中的数据表。

添加一个实体后, Xcode 界面的 ENTITIES 目录中会自动生成一个命名为 Entity 的实体, 在 Default 目录中可以重新设置这个实体的名称为 Student, 操作步骤如图 9-10 所示。

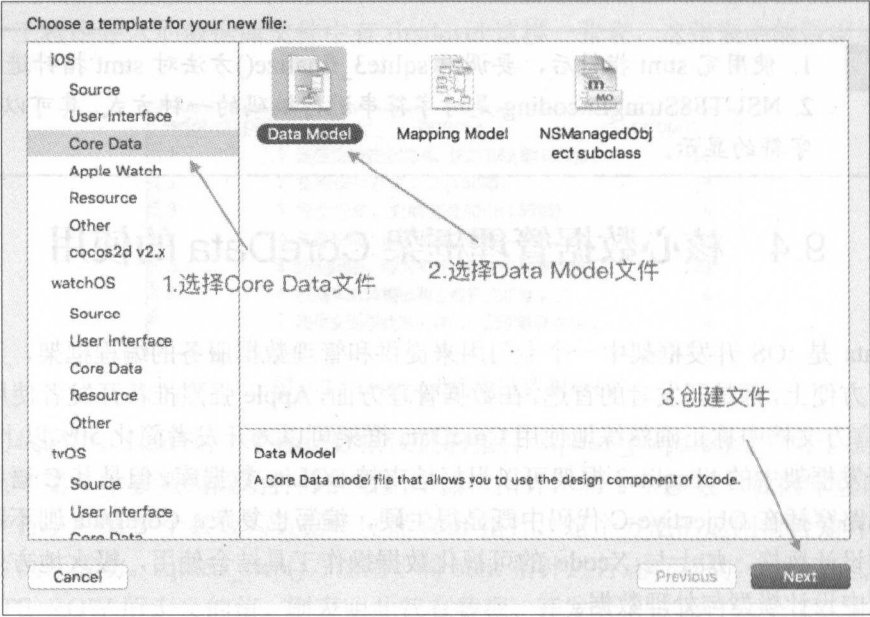


图 9-9 创建 CoreData 数据模型文件

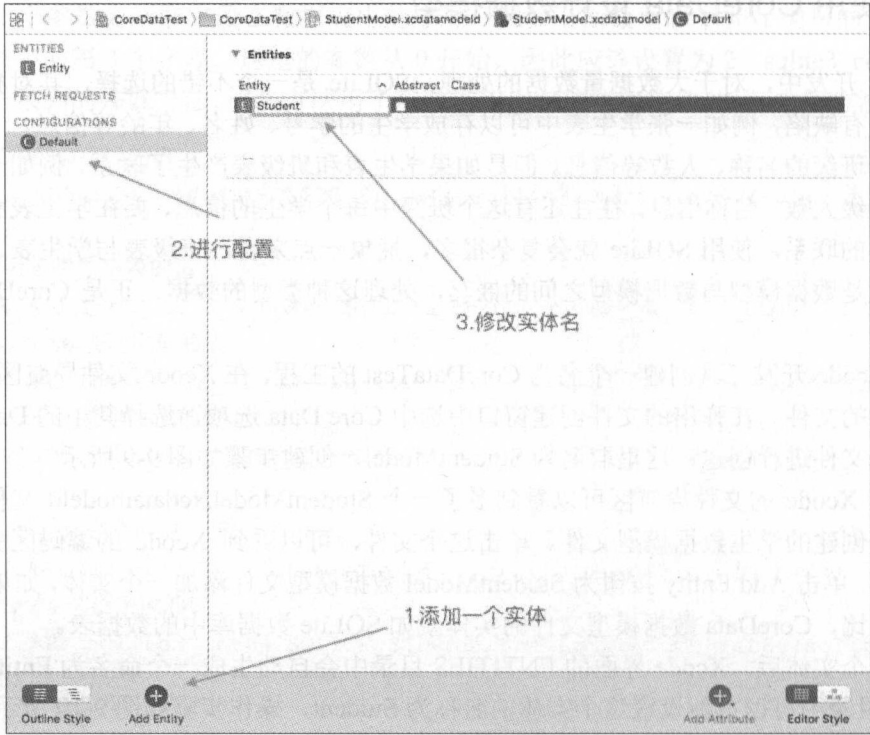


图 9-10 添加并修改实体名

下面为 Student 实体添加几个属性，类似于向数据库表中添加字段一样，单击 Xcode 实体目录中的 Student 实体，在 Attributes 属性栏中为其添加两个属性，一个代表学生名称的 name 属性，一个代表学号的 number 属性，步骤如图 9-11 所示。

以同样的方式再创建一个名为 `SchoolClass` 的实体，作为班级的数据模型，为 `SchoolClass` 实体添加两个属性，一个 `count` 代表学生人数，一个 `name` 代表班级名称，如图 9-12 所示。

实体之间有时并不独立，例如代表班级的 `SchoolClass` 实体中可能会有一个班长的属性，而班长就是 `Student` 实体类型。这样，数据模型和数据模型之间就很容易会产生复杂的关系，在 `CoreData` 中，使用 `Relationships` 来描述实体之间的关系，选择 `SchoolClass` 实体，在 `Relationships` 一栏单击加号新建一个关系，将其命名为 `monitor`，代表此班级中的班长，将 `monitor` 的 `Destination` 选择为 `Student` 实体，如图 9-13 所示。



图 9-11 向实体中添加属性

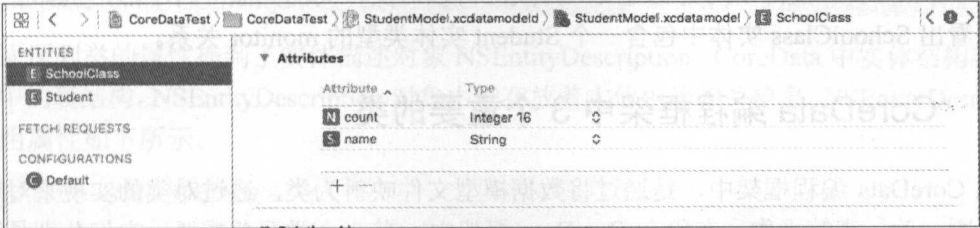


图 9-12 添加一个 `SchoolClass` 实体

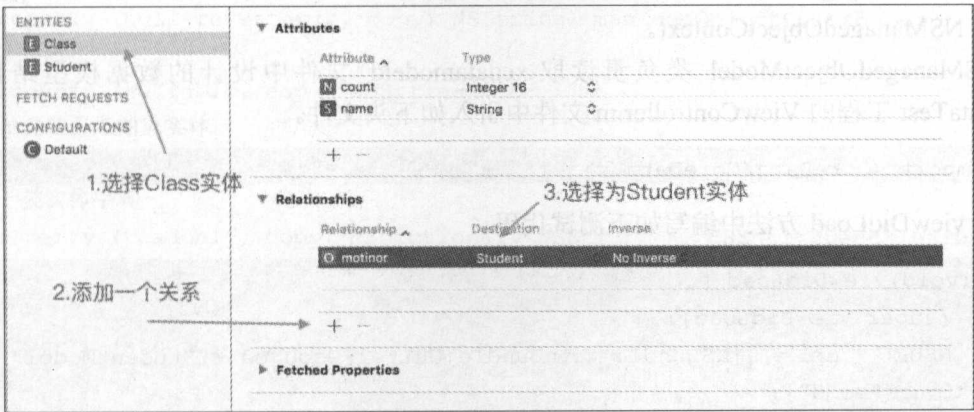


图 9-13 在实体间建立关系

`Xcode` 开发工具还提供了一种更加直观的编辑方式来设置实体的属性和之间的关系。在 `CoreData` 数据模型设置界面的右下角有一个切换编辑模式的按钮提供给开发者进行编辑模式的切换。如图 9-14 所示。

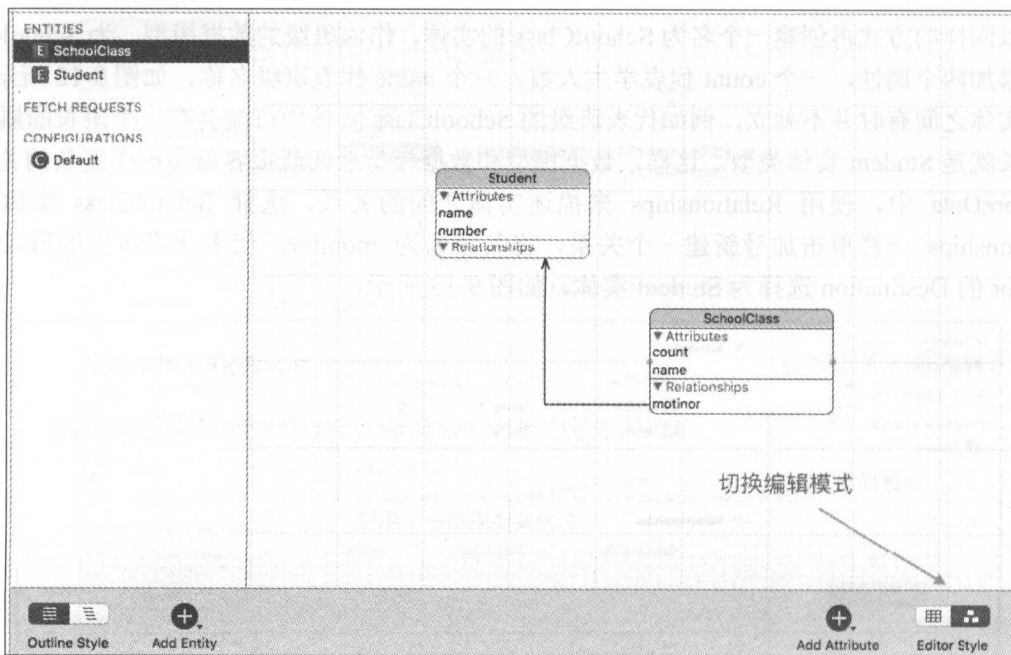


图 9-14 切换编辑模式

图 9-14 所示的编辑模式更加直观，开发者可以对实体间的关系一目了然，例如上面很容易可以看出 SchoolClass 实体中包含一个 Student 实体类型的 monitor 关系。

9.4.2 CoreData 编程框架中 3 个重要的类

在 CoreData 编程框架中，是通过将数据模型文件映射为类，通过对类的实现来对数据进行增、删、改、查等操作。在整个 CoreData 框架中，有 3 个类至关重要，它们分别是数据模型管理类 `NSManagedObjectModel`、持久化数据桥接类 `NSPersistentStoreCoordinator`、数据操作上下文 `NSManagedObjectContext`。

`NSManagedObjectModel` 类负责读取 `.xcdatamodeld` 文件中设计的数据模型结构，在 CoreDataTest 工程的 `ViewController.m` 文件中引入如下头文件。

```
#import <CoreData/CoreData.h>
```

在 `viewDidLoad` 方法中编写如下测试代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSURL * url = [[NSBundle mainBundle] URLForResource:@"StudentModel" withExtension:@"momd"];
    NSManagedObjectModel * model = [[NSManagedObjectModel alloc] initWithContentsOfURL:url];
    NSLog(@"%@", model.entitiesByName);
}
```

上面代码中将.xcdatamodeld 文件中的数据结构读取为 NSManagedObjectModel 对象。NSManagedObjectModel 类中常用的方法和属性列举如下所示。

```
//将多个数据模型管理文件进行合并
+ (nullable NSManagedObjectModel *)mergedModelFromBundles:(nullable NSArray<NSBundle *> *)bundles;

//将多个数据模型管理类对象进行合并
+ (nullable NSManagedObjectModel *)modelByMergingModels:(nullable NSArray<NSManagedObjectModel *> *)models;

//存放数据中所有实体模型的字典 字典中是实体名和实体描述对象
@property (readonly, copy) NSDictionary<NSString *, NSEntityDescription *> *entitiesByName;

//存放数据中所有实体描述对象
@property (strong) NSArray<NSEntityDescription *> *entities;

//返回所有可用的配置名称
@property (readonly, strong) NSArray<NSString *> *configurations;

//创建数据操作请求模板
- (void)setFetchRequestTemplate:(nullable NSFetchRequest *)fetchRequestTemplate forName:(NSString *)name;

//获取数据操作请求模板
- (nullable NSFetchRequest *)fetchRequestTemplateForName:(NSString *)name;
```

上面所列举的属性提到了实体描述对象 NSEntityDescription, CoreData 中实体结构类似于数据库中的表结构, NSEntityDescription 对象中就存放着实体中的相关信息, NSEntityDescription 类中常用属性如下所示。

```
//实体所在的模型管理对象
@property (readonly, assign) NSManagedObjectModel *managedObjectModel;

//实体所在的模型管理对象的名称
@property (null_resettable, copy) NSString *managedObjectClassName;

//实体名
@property (nullable, copy) NSString *name;

//设置是否是抽象实体
@property (getter=isAbstract) BOOL abstract;

//子类实体字典
@property (readonly, copy) NSDictionary<NSString *, NSEntityDescription *> *subentitiesByName;

//所有子类实体对象数组
@property (strong) NSArray<NSEntityDescription *> *subentities;

//父类实体
@property (nullable, readonly, assign) NSEntityDescription *superentity;

//所有属性字典
@property (readonly, copy) NSDictionary<NSString *, __kindof NSPropertyDescription *> *propertiesByName;

//所有属性数组
```

```

@property (strong) NSArray<__kindof NSPropertyDescription *> *properties;
//所有常类型属性
@property (readonly, copy) NSDictionary<NSString *, NSAttributeDescription
n *> *attributesByName;
//所有关系
@property (readonly, copy) NSDictionary<NSString *, NSRelationshipDescription
tion *> *relationshipsByName;
//某个实体类型的所有关系
- (NSArray<NSRelationshipDescription *> *)relationshipsWithDestinationEnt
ity:(NSEntityDescription *)entity;
//判断是否是某种实体
- (BOOL)isKindOfEntity:(NSEntityDescription *)entity;

```

上面列举的属性中提到了 `NSAttributeDescription` 类和 `NSRelationshipDescription` 类，这两个类都继承于 `NSPropertyDescription` 类，用于描述实体中的属性和关系信息，不同的是，`NSAttributeDescription` 描述基本数据类型的属性，`NSRelationshipDescription` 用于描述自定义数据模型类型的关系。

`NSPersistentStoreCoordinator` 建立数据模型与本地文件或数据库之间的联系，通过它将本地数据读入内存或者将修改过的临时数据进行持久化的保存。其初始化与链接数据持久化接收对象方法如下：

```

//通过数据模型管理对象进行初始化
- (instancetype)initWithManagedObjectModel:(NSManagedObjectModel *)model;
//添加一个持久化的数据接收对象
- (nullable __kindof NSPersistentStore *)addPersistentStoreWithType:(NSString
*)storeType configuration:(nullable NSString *)configuration URL:(nullable
NSURL *)storeURL options:(nullable NSDictionary *)options error:(NSError *
*)error;
//移除一个持久化的数据接收对象
- (BOOL)removePersistentStore:(NSPersistentStore *)store error:(NSError *
*)error;

```

`NSManagedObjectContext` 类是进行数据管理的核心类，开发者通过这个类来进行数据的增删改查等操作。其中常用方法如下所示。

```

//初始化方法 参数枚举如下：
/*
typedef NS_ENUM(NSUInteger, NSManagedObjectContextConcurrencyType) {
    NSPrivateQueueConcurrencyType      = 0x01, //上下文对象与私有队列关联
    NSMainQueueConcurrencyType         = 0x02 //上下文对象与主队列关联
};
*/
- (instancetype)initWithConcurrencyType:(NSManagedObjectContextConcurren
cyType)ct;
//异步执行 block 方法块

```



```

- (void)performBlock:(void (^)(void))block;
//同步执行 block 方法块
- (void)performBlockAndWait:(void (^)(void))block;
//关联数据持久化对象
@property (nullable, strong) NSPersistentStoreCoordinator *persistentStoreCoordinator;
//是否有未提交的更改
@property (nonatomic, readonly) BOOL hasChanges;
//进行查询数据请求
- (nullable NSArray *)executeFetchRequest:(NSFetchRequest *)request error:(NSError **)error;
//进行查询数据条数请求
- (NSUInteger) countForFetchRequest:(NSFetchRequest *)request error:(NSError **)error ;
//插入元素
- (void)insertObject:(NSManagedObject *)object;
//删除元素
- (void)deleteObject:(NSManagedObject *)object;
//回滚一步操作
- (void)undo;
//清除缓存
- (void)reset;
//还原数据
- (void)rollback;
//提交保存数据
- (BOOL)save:(NSError **)error;

```

本小节主要向读者独立介绍了 CoreData 框架中核心的管理类的基础概念与属性方法，下一小节将综合使用这些类来进行数据的交互操作。

9.4.3 CoreData 编程框架的数据操作

CoreData 编程框架中一系列对数据模型的映射、持久化协调者的关联和数据模型操作对象的实例化等过程十分复杂，但是其面向开发者进行真正的数据操作时就会方便许多。CoreData 框架中各个模块的协调合作过程如图 9-15 所示。

图 9-15 中，.xcdatamodeld 数据模型文件除了映射为 NSManagedObjectModel 数据模型管理类之外，其还会将他的所有实体映射为具体的数据模型类，类如学生类，班级类等。通过 NSPersistentStoreCoordinator 协调类将 NSManagedObjectModel 类与本地的数据库或文件进行结合，为数据提供持久化的支持。NSManagedObjectContext 是面向开发者的操作上下文类，通过它，开发者可以进行数据操作，通过这一系列的映射与关联，开发者实际上间接操作了具体数据模型实体对象。

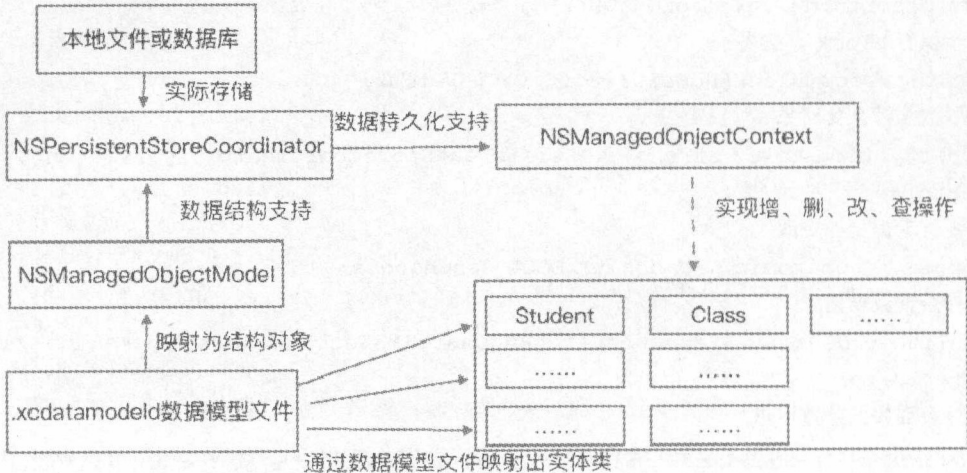


图 9-15 CoreData 框架中各模块协调合作示意图

打开上一小节创建的 CoreDataTest 工程，首先需要将 StudentModel 模型文件中的实体映射为具体的数据模型类，这个过程可以叫作数据模型的类化。选中工程中的 StudentModel.xcdatamodeld 文件，单击 Xcode 工具菜单栏中的 Editor 选项，单击 Create NSManagedObject Subclass 选项，如图 9-16 所示。

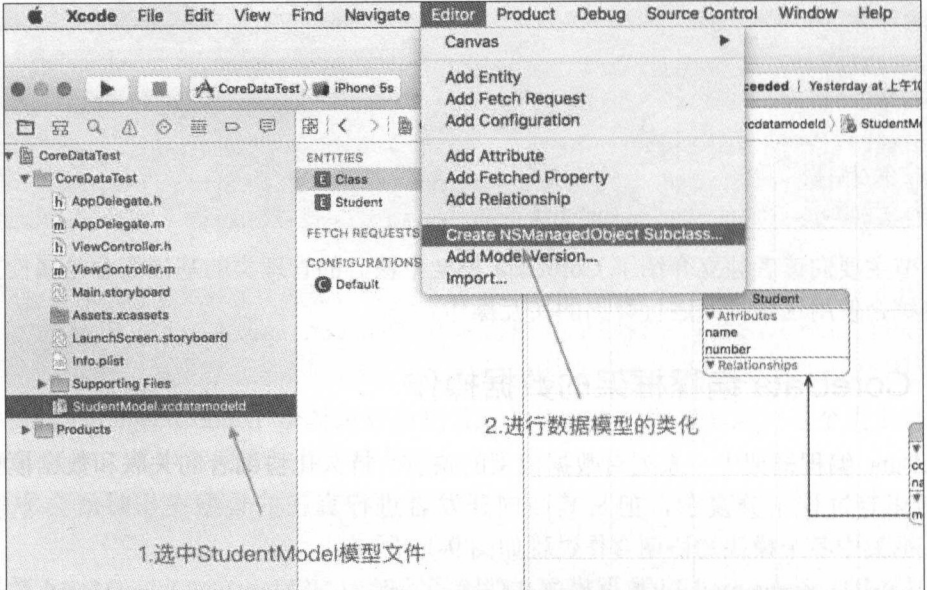


图 9-16 将数据模型映射为类

在之后弹出的模型文件选择窗口中勾选 StudentModel 文件，如图 9-17 所示。

在弹出的要进行类化的实体选择窗口中，选择 Student 实体和 SchoolClass 实体，如图 9-18 所示。

之后，Xcode 工程中会多出 8 个文件，这些文件中进行了相应类的属性声明。

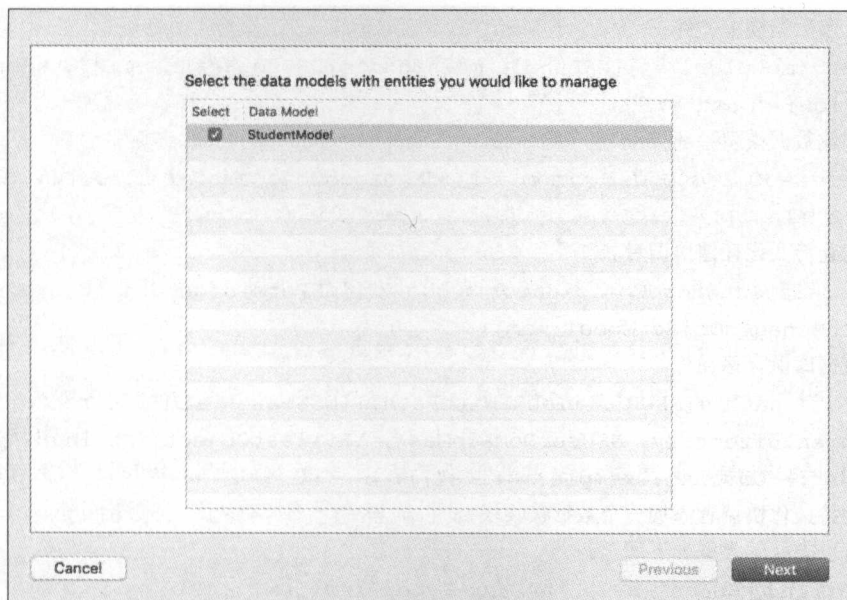


图 9-17 勾选要类化的模型文件

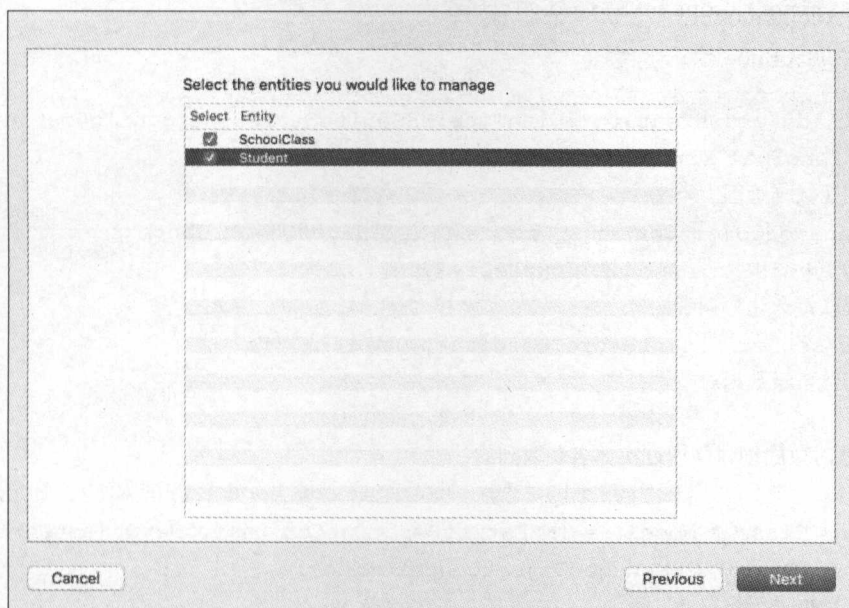


图 9-18 勾选要进行类化的实体

在 ViewController.m 文件中引入如下的头文件。

```
#import "SchoolClass.h"
#import "Student.h"
```

在 viewDidLoad 方法中实现如下代码进行数据的创建操作。

```
- (void)viewDidLoad {
    [super viewDidLoad];
```



```

//读取数据模型文件
NSURL *modelUrl = [[NSBundle mainBundle] URLForResource:@"StudentModel"
withExtension:@"momd"];
//创建数据模型管理类
NSManagedObjectModel * mom = [[NSManagedObjectModel alloc] initWithCont
entsOfURL:modelUrl];
//创建持久化存储协调者
NSPersistentStoreCoordinator * psc = [[NSPersistentStoreCoordinator al
loc] initWithManagedObjectModel:mom];
//数据库保存路径
NSURL * path = [NSURL fileURLWithPath:[NSSearchPathForDirectoriesInDom
ains(NSDocumentDirectory, NSUserDomainMask, YES) lastObject] stringByAppending
PathComponent:@"CoreDataExample.sqlite"]];
//为持久化协调者添加一个数据接收栈
/*可以支持的类型如下所示。
NSString * const NSSQLiteStoreType; //sqlite
NSString * const NSXMLStoreType; //XML
NSString * const NSBinaryStoreType; //二进制
NSString * const NSInMemoryStoreType; //内存
*/
[psc addPersistentStoreWithType:NSSQLiteStoreType configuration:nil U
RL:path options:nil error:nil];
//创建数据管理上下文
NSManagedObjectContext * moc = [[NSManagedObjectContext alloc] initWith
ConcurrencyType:NSMainQueueConcurrencyType];
//关联持久化协调者
[moc setPersistentStoreCoordinator:psc];
//创建班级数据对象
/*
数据对象的创建是通过实体名获取到的
*/
SchoolClass * modelS = [NSEntityDescription insertNewObjectForEntityFo
rName:@"SchoolClass" inManagedObjectContext:moc];
//对数据进行设置
modelS.name = @"第一班";
modelS.count = @60;
//创建学生数据对象
Student * modelSt = [NSEntityDescription insertNewObjectForEntityForNa
me:@"Student" inManagedObjectContext:moc];
modelSt.name = @"张三";
modelSt.number = @"14231233";
modelS.monitor = modelSt;
//进行存储

```

```

    if ([moc save:nil]) {
        NSLog(@"新增成功");
    }
    //打印创建的数据库路径
    NSLog(@"%@", [[NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
    NSUserDomainMask, YES)lastObject] stringByAppendingPathComponent:@"CoreDataE
xample.sqlite"]);
}

```

上面的代码演示了 CoreData 进行数据操作并进行数据持久化存储的完整过程，最后打印出的地址为模拟器生成的数据库文件的地址，前往地址所在的文件夹，读者应该可以看到生成的 SQLite 数据库文件。

CoreData 通过 NSFetchedRequest 查询请求类来进行数据查询的相关操作，NSFetchedRequest 类用于创建一个查询请求，其中常用方法与属性如下所示。

```

//创建一个实体的查询请求 可以理解为在某个表中进行查询
+ (instancetype)fetchRequestWithEntityName:(NSString*)entityName;
//查询条件
@property (nullable, nonatomic, strong) NSPredicate *predicate;
//数据排序
@property (nullable, nonatomic, strong) NSArray<NSSortDescriptor *> *sort
Descriptors;
//每次查询返回的数据条数
@property (nonatomic) NSUInteger fetchLimit;
//设置查询到数据的返回类型
@property (nonatomic) NSFetchedRequestResultType resultType;
//设置查询结果是否包含子实体
@property (nonatomic) BOOL includesSubentities;
//设置要查询的属性值
@property (nullable, nonatomic, copy) NSArray *propertiesToFetch;

```

例如，查询上面添加的 SchoolClass 模型数据，使用如下的代码。

```

//创建一条查询请求
NSFetchedRequest * request = [NSFetchedRequest fetchRequestWithEntityName:
@"SchoolClass"];
//设置条件为 count=60 的数据
[request setPredicate:[NSPredicate predicateWithFormat:@"count == 60"]];
//进行查询操作
NSArray * res = [moc executeFetchRequest:request error:nil];
NSLog(@"班长:%@ %@ ,班级:%@", (Student *) [res.firstObject monitor].name, (St
udent *) [res.firstObject monitor].number, [res.firstObject name]);

```

运行工程，可以看到在 Xcode 的调试区打印出查询到的信息，这跟前边添加的数据一致，如图 9-19 所示。

2016-04-05 21:39:55.152 CoreDataTest[1059:50320]
班长:张三 14231233 ,班级:第一班

图 9-19 Xcode 调试区的打印数据

9.4.4 使用 CoreData 进行数据与页面的绑定

CoreData 编程框架不只可以抽象地操作数据，其更加强大的地方在于提供了视图与数据的绑定功能，通过 `NSFetchedResultsController` 类，开发者可以十分方便地进行数据与页面的关联与绑定。



提示

所谓页面与数据的绑定是指数据不仅显示在页面上，数据与页面也有双向的关联，当数据变化时，页面可以同步的刷新；同样，当页面显示变化时，数据也要进行同步的更新。

使用 Xcode 创建一个名为 `CoreDataModelView` 的工程，用于演示使用 CoreData 框架进行数据与页面的绑定。在工程中创建一个 CoreData 数据模型文件，取名为 `StudentModel`，在 `StudentModel` 模型文件中添加一个 `Student` 实体，为其添加两个属性，分别取名为 `name` 与 `age`，代表学生的名字和年龄，如图 9-20 所示。

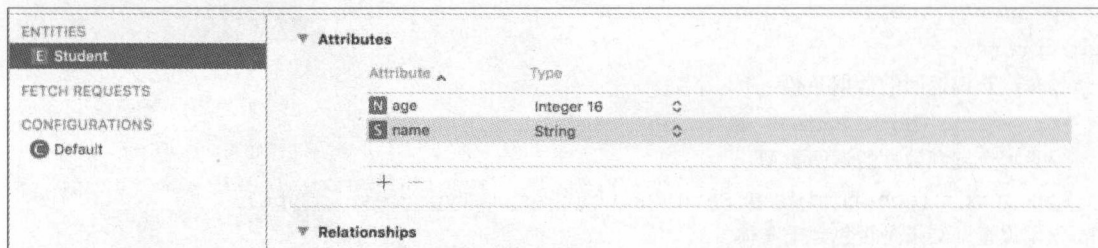


图 9-20 在数据模型文件中创建 Student 实体

使用前面小节介绍的方法添加两条数据，首先进行数据模型的类化，之后在 `ViewController.m` 文件中引入相应的头文件，最后在 `viewDidLoad` 方法中实现如下示例代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    //读取数据模型文件
    NSURL *modelUrl = [[NSBundle mainBundle] URLForResource:@"Student" with
Extension:@"momd"];
    //创建数据模型
    NSManagedObjectModel * mom = [[NSManagedObjectModel alloc] initWithCont
entsOfURL:modelUrl];
    //创建持久化存储协调者
```



```

    NSPersistentStoreCoordinator * psc = [[NSPersistentStoreCoordinator alloc] initWithManagedObjectModel:mom];
    //数据库保存路径
    NSURL * path = [NSURL fileURLWithPath:[NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES)lastObject] stringByAppendingPathComponent:@"CoreDataExample.sqlite"];
    //为持久化协调者添加一个数据接收栈
    /*可以支持的类型如下所示。
    NSString * const NSSQLiteStoreType;//sqlite
    NSString * const NSXMLStoreType;//XML
    NSString * const NSBinaryStoreType;//二进制
    NSString * const NSInMemoryStoreType;//内存
    */
    [psc addPersistentStoreWithType:NSSQLiteStoreType configuration:nil URL:path options:nil error:nil];
    //创建数据管理上下文
    NSManagedObjectContext * moc = [[NSManagedObjectContext alloc] initWithConcurrencyType:NSMainQueueConcurrencyType];
    //关联持久化协调者
    [moc setPersistentStoreCoordinator:psc];
    //创建数据对象
    /*
    数据对象的创建是通过实体名获取到的
    */
    Student * modelS = [NSEntityDescription insertNewObjectForEntityForName:@"Student" inManagedObjectContext:moc];
    //对数据进行设置
    modelS.name = @"张三";
    modelS.age = @16;
    Student * modelS2 = [NSEntityDescription insertNewObjectForEntityForName:@"Student" inManagedObjectContext:moc];
    //对数据进行设置
    modelS2.name = @"李四";
    modelS2.age = @17;
    //进行存储
    if ([moc save:nil]) {
        NSLog(@"新增成功");
    }
}

```

确定添加数据成功后, 将 viewDidLoad 方法中的代码删去, 下面将进行数据与页面绑定的代码演示。

在 iOS 应用开发中, 实际上大多数大数据量的展示都是由 UITableView 完成的, 在

ViewController.m 文件中遵守如下协议并添加一个 NSFetchedResultsController 类型的属性，如下所示。

```
//遵守协议
@interface ViewController ()<NSFetchedResultsControllerDelegate, UITableViewDataSource, UITableViewDelegate>
{
    //数据桥接对象
    NSFetchedResultsController * _fecCon;
}
@end
```

在 viewDidLoad 方法中进行如下设置操作。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    //进行初始化操作
    NSURL *modelUrl = [[NSBundle mainBundle] URLForResource:@"StudentModel"
withExtension:@"momd"];
    NSManagedObjectModel * mom = [[NSManagedObjectModel alloc] initWithContentsOfURL:modelUrl];
    NSPersistentStoreCoordinator * psc = [[NSPersistentStoreCoordinator alloc] initWithManagedObjectModel:mom];
    NSURL * path =[NSURL fileURLWithPath:[NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES)lastObject] stringByAppendingPathComponent:@"CoreDataExample.sqlite"];
    [psc addPersistentStoreWithType:NSSQLiteStoreType configuration:nil URL:path options:nil error:nil];
    NSManagedObjectContext * moc = [[NSManagedObjectContext alloc] initWithConcurrencyType:NSMainQueueConcurrencyType];
    [moc setPersistentStoreCoordinator:psc];
    NSFetchedRequest * request = [NSFetchedRequest fetchRequestWithEntityName:@"Student"];
    //设置数据排序
    [request setSortDescriptors:@[[NSSortDescriptor sortDescriptorWithKey:@"age" ascending:YES]]];
    //进行数据绑定对象的初始化
    _fecCon = [[NSFetchedResultsController alloc] initWithFetchRequest:request managedObjectContext:moc sectionNameKeyPath:nil cacheName:nil];
    //设置代理
    _fecCon.delegate=self;
    //进行数据查询
    [_fecCon performFetch:nil];
    //界面初始化
```

```

UITableView * tableView = [[UITableView alloc] initWithFrame:self.view.
bounds style:UITableViewStylePlain];
[self.view addSubview:tableView];
tableView.delegate = self;
tableView.dataSource = self;
}

```

在 ViewController.m 文件中实现 UITableView 控件的代理方法如下所示。

```

-(UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
(NSIndexPath *)indexPath{
    UITableViewCell * cell = [tableView dequeueReusableCellWithIdentifier:
@"cellid"];
    if (!cell) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyle
ubtitle reuseIdentifier:@"cellid"];
    }
    //获取相应数据模型
    Student * obj = [_fecCon objectAtIndex:indexPath:indexPath];
    cell.textLabel.text = obj.name;
    cell.detailTextLabel.text = [NSString stringWithFormat:@"年龄: %@",obj.
age];
    return cell;
}

-(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView{
    //获取数据绑定对象中数据分区个数
    return [_fecCon sections].count;
}

-(NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSI
nteger)section{
    //获取数据分区信息
    id<NSFetchedResultsSectionInfo> info = [_fecCon sections][section];
    //返回分区行数
    return [info numberOfObjects];
}

```

运行工程，效果如图 9-21 所示。

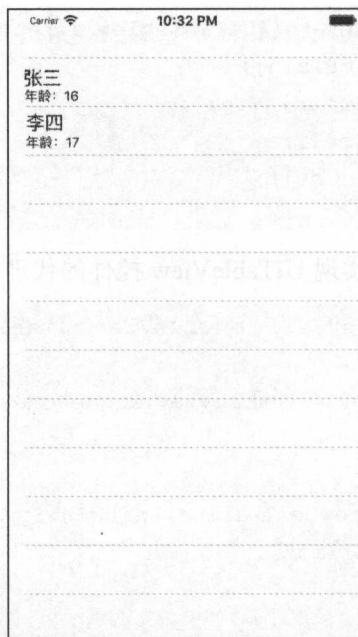


图 9-21 在 UITableView 控件上实现 CoreData 数据绑定

当数据库中的数据进行了修改之后，会执行一系列的代理方法，在 ViewController.m 文件中实现这些方法如下所示。

```
//数据将要改变时调用的方法
- (void)controllerWillChangeContent:(NSFetchedResultsController *)controller
{
    //开启 tableView 更新预处理
    [[self tableView] beginUpdates];
}

//分区数据改变时调用的方法
- (void)controller:(NSFetchedResultsController *)controller didChangeSection:
(id <NSFetchedResultsSectionInfo>)sectionInfo atIndex:(NSUInteger)sectionIndex
forChangeType:(NSFetchedResultsChangeType)type
{
    //判断行为类型
    switch (type) {
        //插入新分区
        case NSFetchedResultsChangeInsert:
            [[self tableView] insertSections:[NSIndexSet indexSetWithIndex:
sectionIndex] withRowAnimation:UITableViewRowAnimationFade];
            break;
        //删除分区
        case NSFetchedResultsChangeDelete:
            [[self tableView] deleteSections:[NSIndexSet indexSetWithIndex:
sectionIndex] withRowAnimation:UITableViewRowAnimationFade];
            break;
    }
}
```

```

        break;
    //移动分区
    case NSFetchedResultsControllerChangeMove:
    //更新分区
    case NSFetchedResultsControllerChangeUpdate:
        break;
    }
}
//数据改变时回调的代理
- (void)controller:(NSFetchedResultsController *)controller didChangeObject:(id)anObject atIndexPath:(NSIndexPath *)indexPath forChangeType:(NSFetchedResultsControllerChangeType)type newIndexPath:(NSIndexPath *)newIndexPath
{
    switch(type) {
        //插入数据
        case NSFetchedResultsControllerChangeInsert:
            [[self tableView] insertRowsAtIndexPaths:@[newIndexPath] withRowAnimation:UITableViewRowAnimationFade];
            break;
        //删除数据
        case NSFetchedResultsControllerChangeDelete:
            [[self tableView] deleteRowsAtIndexPaths:@[indexPath] withRowAnimation:UITableViewRowAnimationFade];
            break;
        //更新数据
        case NSFetchedResultsControllerChangeUpdate:
            [self reloadData];
            break;
        //移动数据
        case NSFetchedResultsControllerChangeMove:
            [[self tableView] deleteRowsAtIndexPaths:@[indexPath] withRowAnimation:UITableViewRowAnimationFade];
            [[self tableView] insertRowsAtIndexPaths:@[newIndexPath] withRowAnimation:UITableViewRowAnimationFade];
            break;
    }
}
//数据更新结束调用的代理
- (void)controllerDidChangeContent:(NSFetchedResultsController *)controller
{
    [[self tableView] endUpdates];
}

```

实现了如上的代理方法，当与页面绑定的数据发生变化时，页面会同步进行刷新。

9.5 网络缓存策略

网络缓存模块是移动网络应用的核心模块,对于一些固定的数据或者实时性要求并不太强的数据开发者往往会在应用中使用缓存策略来对数据进行持久化保存直到缓存的数据过期,在缓存过期之前,相同的网络请求都将被拦截。这种做法有许多优势,列举如下。

- (1) 节省用户网络传输流量。
- (2) 本地数据可以更快地加载,优化用户体验。
- (3) 在断网的情况下,用户依然可以浏览曾经浏览过的数据。

9.5.1 为网络请求设置缓存策略

使用 Xcode 创建一个名为 RequestCacheTest 的工程,首先在工程的 Info.plist 文件中添加支持工程进行 HTTP 请求的相关键值。在 ViewController.m 文件的 viewDidLoad 方法中添加如下测试代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSURL * url = [NSURL URLWithString:@"http://www.baidu.com"];
    NSURLRequest * request = [NSURLRequest requestWithURL:url cachePolicy:
    NSURLRequestReturnCacheDataElseLoad timeoutInterval:10];
    [NSURLConnection sendAsynchronousRequest:request queue:[NSOperationQu
    eue mainQueue] completionHandler:^(NSURLResponse * _Nullable response, NSData
    * _Nullable data, NSError * _Nullable connectionError) {
        NSLog(@"%@",data);
    }];
}
```

上面代码中,使用 NSURLConnection 的类方法从 www.baidu.com 地址进行数据请求,这里的 sendAsynchronousRequest:queue:completionHandler:方法通过异步的方式进行网络请求,请求返回的数据在在 block 中通过参数传递给开发者。在创建 NSURLRequest 请求对象时,上面使用了 requestWithURL:cachePolocy:timeoutInterval:方法,这个方法第 1 个参数设置请求地址 URL,第 2 个参数是请求的缓存策略,第 3 个参数设置请求的超时时间,测试代码中设置了 10s,如果 10s 后无响应数据,则请求会按因超时失败处理。缓存策略可以设置的参数为 NSURLRequestCachePolicy 类型的枚举,这个枚举中除了一些为定义的缓存策略外,常用策略如下所示。

```
typedef NS_ENUM(NSUInteger, NSURLRequestCachePolicy)
{
    //使用 HTTP/HTTPS 协议中定义的缓存策略
    NSURLRequestUseProtocolCachePolicy = 0,
```



```
//无论有无本地缓存数据，都从服务器进行请求
NSURLRequestReloadIgnoringLocalCacheData = 1,
//先检查缓存数据，若无缓存再进行请求
NSURLRequestReturnCacheDataElseLoad = 2,
//类似于离线模式 无论有无缓存都不进行请求
NSURLRequestReturnCacheDataDontLoad = 3,
};
```

上面的缓存策略中，最常用的是 `NSURLRequestReturnCacheDataElseLoad`，这种策略也是大多数移动应用的缓存思路。`NSURLRequestUseProtocolCachePolicy` 缓存策略是由服务端定义的，是 HTTP/HTTPS 协议自带的缓存策略。

运行一次上面的工程，之后将网络断掉，再次运行工程，可以看到打印区域依然打印出了请求数据，实际上第 2 次的请求已经不通过网络来获取了，而是直接从本地获取。

9.5.2 应用缓存管理类 `NSURLCache` 简介

在上一小节中介绍了 iOS 网络请求中缓存策略的相关设置，其实缓存策略只是设置了某个请求具体采取的缓存处理方法，缓存区域的存储大小，缓存数据的管理等是由 `NSURLCache` 类来管理的。对于每一个应用程序，系统都默认创建了一个 `NSURLCache` 作为缓存管理对象，开发者可以通过 `NSURLCache` 类的如下方法获取到。

```
//获取当前应用的缓存管理对象
+ (NSURLCache *)sharedURLCache;
```

事实上，开发者也可以通过自定义一个 `NSURLCache` 类对象来作为系统的缓存管理类对象，使用 `NSURLCache` 类的如下类方法。

```
//设置自定义的 NSURLCache 作为应用缓存管理对象
+ (void)setSharedURLCache:(NSURLCache *)cache;
```

可以通过下面的初始化方法来创建自定义的 `NSURLCache` 对象。

```
//初始化一个应用缓存对象
/*
memoryCapacity 设置内存缓存容量
diskCapacity 设置磁盘缓存容量
path 磁盘缓存路径
内容缓存会在应用程序退出后清空 磁盘缓存不会
*/
- (instancetype)initWithMemoryCapacity:(NSUInteger)memoryCapacity diskCapacity:(NSUInteger)diskCapacity diskPath:(nullable NSString *)path;
NSURLCache 类对象的常用方法与属性列举如下：
//获取某一请求的缓存
- (nullable NSCachedURLResponse *)cachedResponseForRequest:(NSURLRequest *)request;
```

```
//给请求设置指定的缓存
- (void)storeCachedResponse:(NSCachedURLResponse *)cachedResponse forRequest:(NSURLRequest *)request;
//移除某个请求的缓存
- (void)removeCachedResponseForRequest:(NSURLRequest *)request;
//移除所有缓存数据
- (void)removeAllCachedResponses;
//移除某个时间起的缓存设置
- (void)removeCachedResponsesSinceDate:(NSDate *)date NS_AVAILABLE(10_10, 8_0);
//内存缓存容量大小
@property NSUInteger memoryCapacity;
//磁盘缓存容量大小
@property NSUInteger diskCapacity;
//当前已用内存容量
@property (readonly) NSUInteger currentMemoryUsage;
//当前已用磁盘容量
@property (readonly) NSUInteger currentDiskUsage;
```

通过上面的方法，开发者可以获取到缓存数据的大小，删除指定请求的缓存数据，删除指定时间点的缓存数据等。

第 10 章

提交应用程序到 AppStore

苹果的官方应用市场只有 AppStore，任何大众应用开发者若要将应用上线供用户使用，都需要将其提交 AppStore 进行审核。在 Xcode 7 之后，开发者可以使用自己免费的 Apple 账号进行真机测试，上线应用依然需要使用具有开发者权限的付费 Apple 账号。

通过本章的学习，读者能够掌握：

1. 对程序执行断点调试的技巧。
2. 申请个人开发者账号
3. 申请公司或企业开发者账号。
4. 使用 Xcode 进行应用程序的打包。
5. 将应用程序提交审核。

10.1 使用 Xcode 开发工具进行程序调试

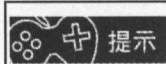
程序的调试与程序的开发有着同样的重要性，任何一个开发者，无论多么优秀多么经验丰富，都不能保证自己编写的代码完全没有 bug。因此，掌握开发中程序调试的知识与技能十分重要。

10.1.1 使用自定义断点进行代码调试

断点是程序调试中最常用的方法，通过断点，开发者可以使程序暂停在所编写的某行代码处，便于检查代码的运行是否存在逻辑错误。使用 Xcode 创建一个名为 BreakPointTest 的工程，在 ViewController.m 文件的 viewDidLoad 方法中添加如下测试代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    int count = 0;
    for (int i=0; i<10; i++) {
        int a = 0;
        int b = a+i;
        count = a+b;
        NSLog(@"%d", count);
    }
}
```

上面的代码十分简单，进行了一些变量的创建和循环操作，向 NSLog() 方法所在的行添加自定义断点，添加断点的方法只需要在 Xcode 左侧行标处单击一下即可，如图 10-1 所示。



如果要删除一个断点，只需将多添加的断点拖动移除即可。

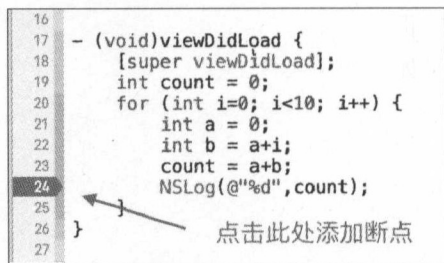


图 10-1 向工程代码中添加断点

此时运行工程，可以看到当执行到断点所在的代码时，程序自动暂停了下来，并且在 Xcode 的调试去将看到当前堆栈块中的相关数据，如图 10-2 所示。

开发者进行断点调试的很多情况都是在循环中，有时并不需要监控每次循环的数据状态，开发者可能只关心某种条件下的循环，例如上面的代码，开发者可以对自定义的断点进行编辑操作，来使其满足一定条件后，程序再暂停，比如设置为当循环变量 i=5 时，再进行断点调试，设置方法如图 10-3 所示。



图 10-2 进行程序断点调试

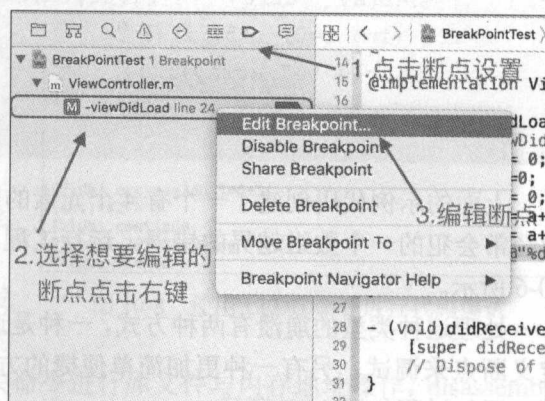


图 10-3 对自定义断点进行编辑

在弹出的断点编辑窗口中，进入如图 10-4 所示的设置。

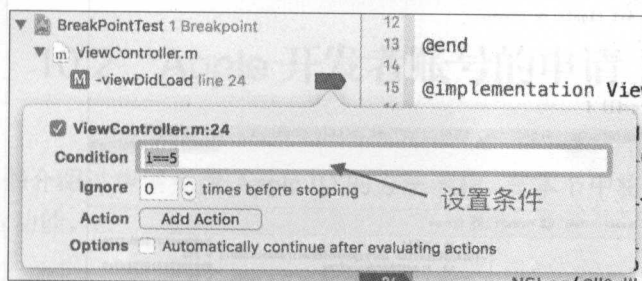


图 10-4 设置断点条件

再次运行工程，可以看到，只有当循环中循环自增变量等于 5 时程序才暂停了下来，其中信息如图 10-5 所示。

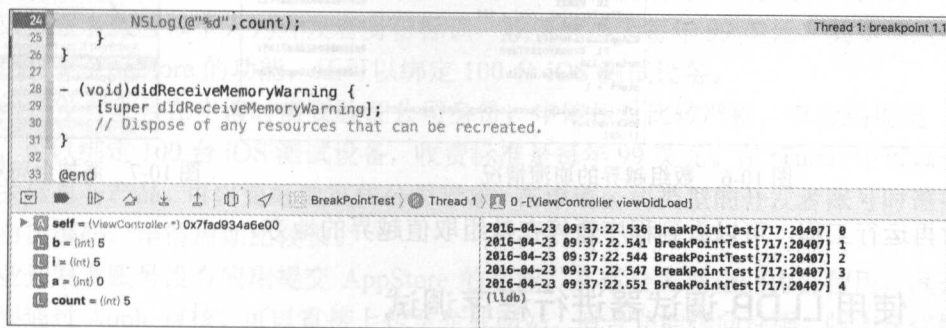


图 10-5 程序调试信息

10.1.2 添加全局异常断点

在开发中还有很多情况是开发者并不清楚是哪里的代码出现了问题，程序直接崩溃在 `main` 函数中，例如可以将 `BreakPointTest` 工程中 `viewDidLoad` 方法中的代码改写如下所示。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    NSArray * array = @[1,2,3,4];
    for (int i=0; i<5; i++) {
        NSLog(@"%",array[i]);
    }
}

```

上面的示例代码创建了一个有 4 个元素的数组但循环打印了 5 次,这是开发者在实际开发中经常会犯的一个数组越界的错误。运行工程,可以看到程序直接崩溃在 main 函数中,如图 10-6 所示。

处理这种类型的崩溃有两种方式,一种是通过打印出的堆栈信息来定位相关代码,添加自定义断点来调试,另有一种更加简单便捷的方式是通过添加全局的异常断点来捕获出错的代码,添加过程如图 10-7 所示。

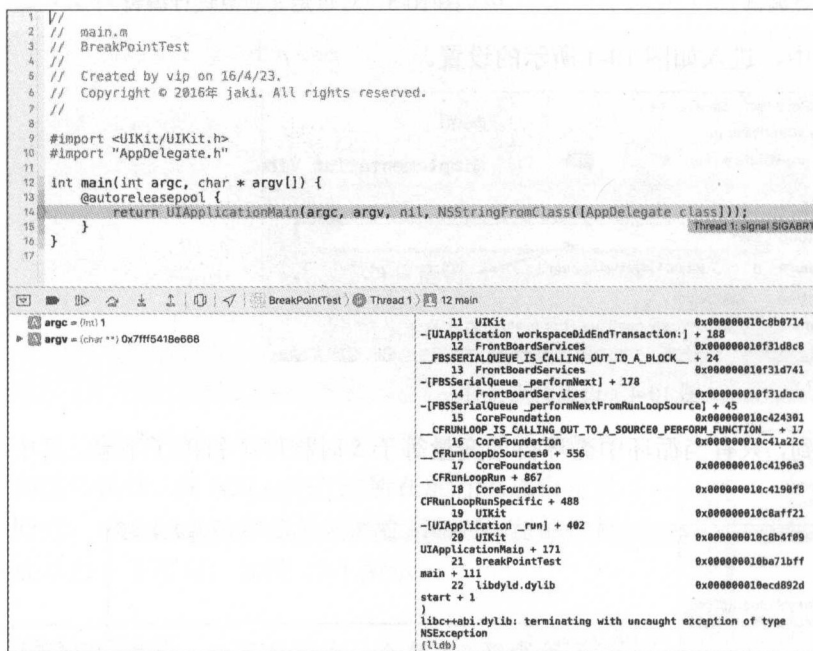


图 10-6 数组越界的崩溃情况

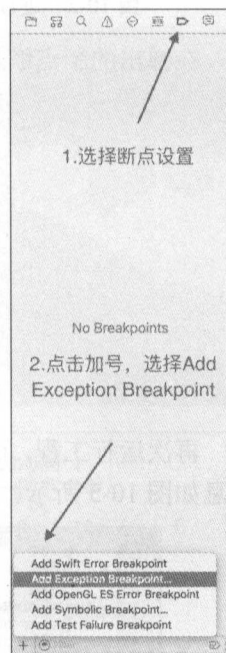


图 10-7 添加全局异常断点

此时再运行工程,可以看到程序暂停在数组取值越界的地方。

10.1.3 使用 LLDB 调试器进行程序调试

LLDB 是 Xcode 开发工具默认的调试工具,其为开发者提供了丰富的代码调试功能,当 Xcode 进行断点 Debug 状态时,开发者可以在调试区输入 LLDB 命令来进行程序调试的相关操作。依然以上一小节添加了全局异常断点的工程来做测试,运行工程,程序断在数组越界的代码处,此时调试区将自动追加加上 lldb 前缀,开发者可以直接通过命令进行调试,图 10-8 所示。

图 10-8 中演示的一些命令是开发中最基础和常用的。例如 `print` 命令用于打印一个对象，它会将对象的类型和地址都打印出来，然而大多数情况下开发者需要查看的是对象具体的值，使用 `po` 命令可以对对象具体的内容进行打印，`expression` 命令用于执行一句代码并将执行结果的返回值打印，例如图 10-8 中示例执行 `i=i+1` 后，当前 `i` 的值变为 5。`continue` 命令为继续线程的运行，`finish` 命令为结束调试，程序继续运行。

上面演示的 LLDB 命令只是冰山一角，开发者还可以使用 `frame` 的相关命令来进行堆栈块的操作，`thread` 相关命令进行线程操作，`image` 相关命令进行库文件与内存地址操作，`disassemble` 命令进行代码反汇编相关操作等，因不是本书重点，这里不再一一介绍，有兴趣的读者可以自行阅读相关资料。

```
2016-04-23 10:33:05.946 BreakPointTest[795:43850] 1
2016-04-23 10:33:05.947 BreakPointTest[795:43850] 2
2016-04-23 10:33:05.947 BreakPointTest[795:43850] 3
2016-04-23 10:33:05.947 BreakPointTest[795:43850] 4
(lldb) print array
(__NSArrayI *) $0 = 0x00007f8e23c04b80 @"4 elements"
(lldb) po array
<__NSArrayI 0x7f8e23c04b80> (
    1,
    2,
    3,
    4
)

(lldb) expression i=i+1
(int) $2 = 5
(lldb) continue
Process 795 resuming
(lldb) finish
```

图 10-8 使用 LLDB 命令进行代码调试

10.2 Apple 开发者账号的申请

在第 1 章中，本书介绍过申请免费 Apple ID 的方法流程，在本节中将介绍如何给 Apple ID 账号申请开通开发者功能。

10.2.1 几种类型的开发者账号

Apple 开发者账号共有 3 种类型，分别为个人开发者账号、公司开发者账号和企业开发者账号。这 3 种账号的年费和功能都会有区别，适合不同性质的开发团队选择使用。

个人开发者账号以个人为开发者身份标识，收费标准为每年 99 美元，其申请速度很快，并有应用提交 AppStore 的功能，还可以绑定 100 台 iOS 测试设备。

公司开发者账号在申请时需要验证公司身份，申请相对比较严格，有应用提交 AppStore 的功能，可以绑定 100 台 iOS 测试设备，收费标准是每年 99 美元。在 iTunes 中可以查询到公司的相关信息与产品，可以更好地宣传公司形象。在申请公司类型的开发者账号时需要提供公司的邓白氏编码，申请周期比较长。

企业开发者账号没有应用提交 AppStore 的功能，其适合大型企业内部使用，其开发的应用不需要通过 Apple 审核，可以直接上传至企业网站，适合功能指向性强，版本迭代快的大型企业使用。企业账号无开发设备的限制，收取的费用为每年 299 美元。

10.2.2 申请开发者账号的过程

个人开发者账号的申请相对较为容易，首先进入 Apple 开发者官网：<https://developer.apple.com/>。进入开发者官网后，界面大致如图 10-9 所示。

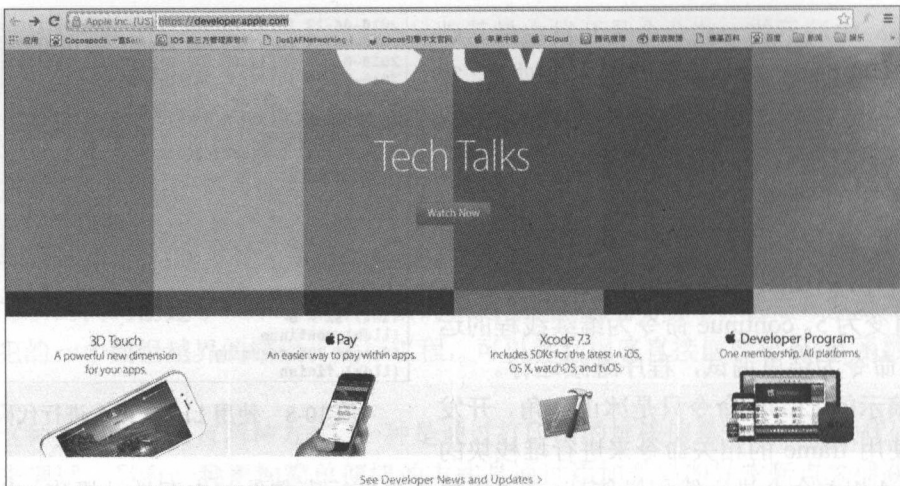


图 10-9 Apple 开发者官网

单击官网中右下方的 **Developer Program** 选项进行界面跳转，之后会进入到 **Apple Developer Program** 界面，界面如图 10-10 所示。

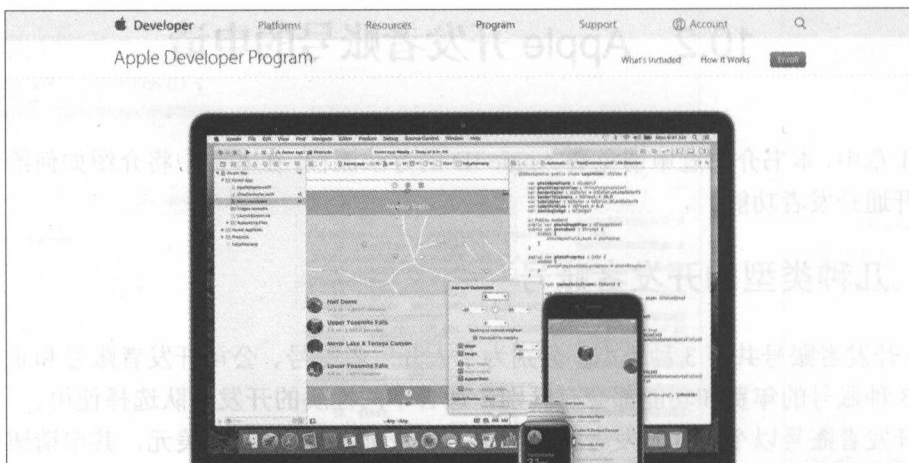


图 10-10 Apple Developer Program 界面

单击 **Apple Developer Program** 界面的 **Enroll** 注册按钮，跳转注册界面，使用第 1 章中所申请的 **Apple ID** 进行登录，之后界面如图 10-11 所示。

单击图 10-11 中的 **Start Your Enrollment** 按钮开始正式注册，界面如图 10-12 所示。

图 10-12 所示界面中的 **I develop apps as** 选项栏用于选择要进行申请的开发者账号类型，其中有 3 个选项，**Individual/Sole Proprietor/Single Person Business** 选项用于申请个人类型的开发者账号，**Company/Organization** 用于申请公司类型的开发者账号，**Government Organization** 用于申请企业类型的开发者账号。首先选择个人类型的开发者账号进行申请操作。单击 **Continue** 继续按钮后，会弹出信息填写界面，如图 10-13 所示。

填写完成个人信息后，单击 **Continue** 继续按钮，之后会再出现一个确认信息的界面，确认无误后单击 **Continue** 按钮继续即可。

在弹出的确认购买界面中确认无误后，单击 **Purchase** 选项进行购买操作，如图 10-14 所示。

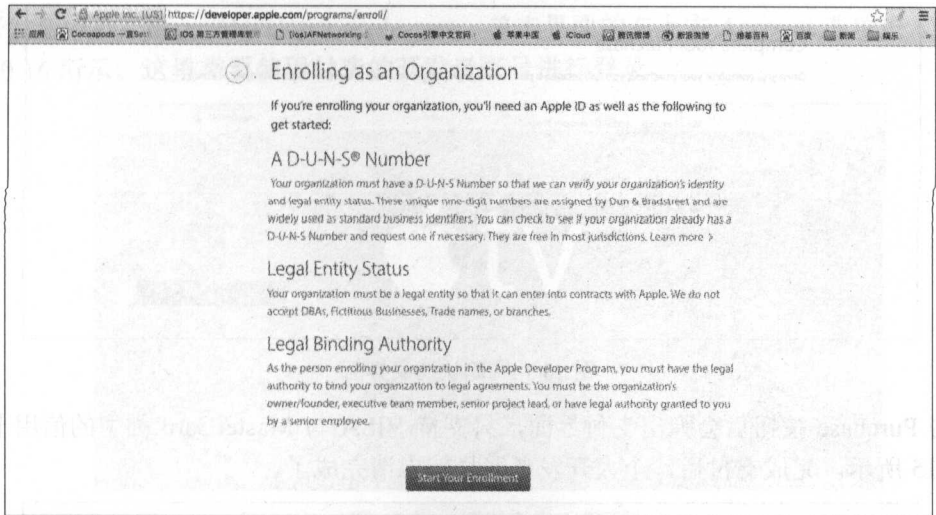


图 10-11 注册开发者账号界面

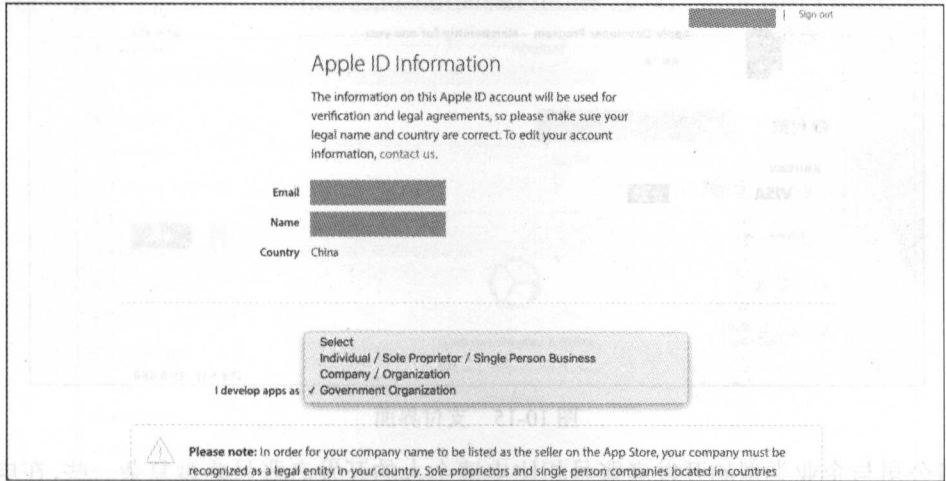


图 10-12 选择希望注册的开发者账号类型

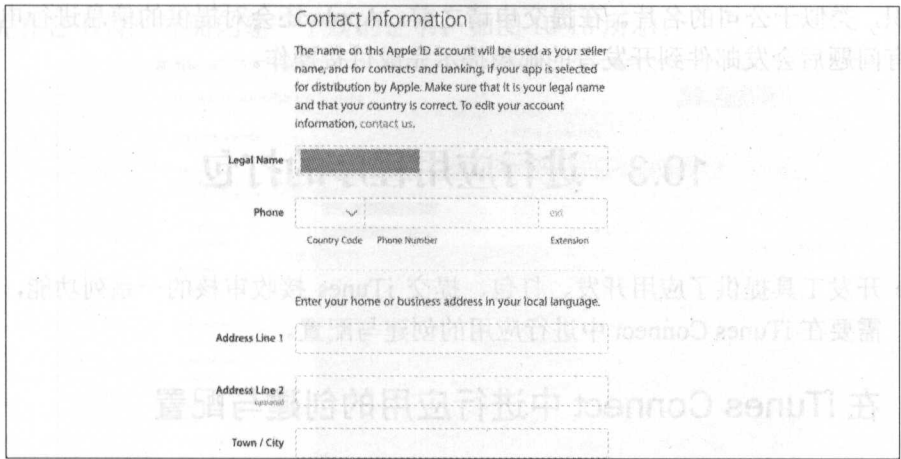


图 10-13 个人信息填写界面

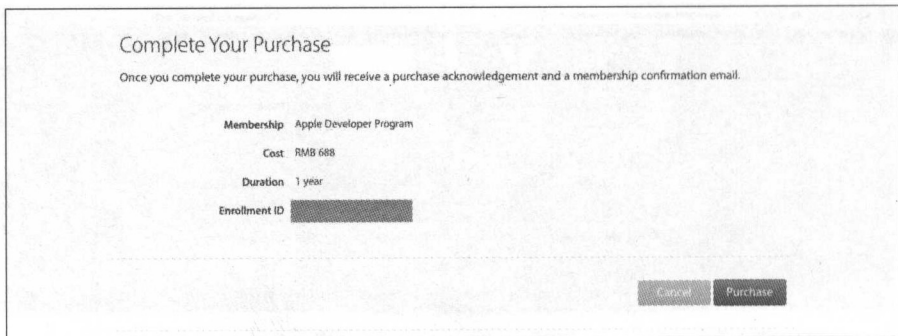


图 10-14 确认购买界面

单击 Purchase 按钮后会弹出支付界面，只支持 VISA 与 MasterCard 类型的信用卡，界面如图 10-15 所示。完成支付后，个人开发者账号就申请完成了。

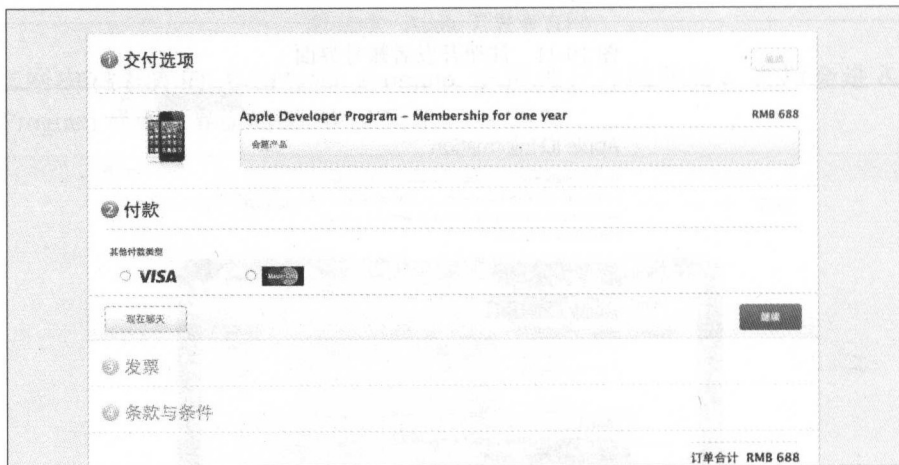


图 10-15 支付界面

申请公司与企业类型的开发者账号相比申请个人的开发者账号略微复杂一些，在申请期间需要上传公司或者企业的相关执照证明，并需要申请一个邓白氏编码，邓白氏编码是国际公司的唯一标识，类似于公司的名片。在提交申请之前，Apple 还会对提供的信息进行电话确认与检核，没有问题后会发邮件到开发者的邮箱提示完成付费操作。

10.3 进行应用程序的打包

Xcode 开发工具提供了应用开发、打包、提交 iTunes 接收审核的一系列功能，在打包提交审核前，需要在 iTunes Connect 中进行应用的创建与配置。

10.3.1 在 iTunes Connect 中进行应用的创建与配置

在 iTunes Connect 中创建应用程序之前，首先需要为要创建的应用程序创建一个套装 ID，

进入开发者中心：<https://developer.apple.com>。单击界面的右上方 Account 选项进入账户管理，如图 10-16 所示。这里需要使用付费的开发者账号进行登录。



图 10-16 开发者中心界面

在账户管理界面选择 Certificates, Identifiers & Profiles 选项，如图 10-17 所示。

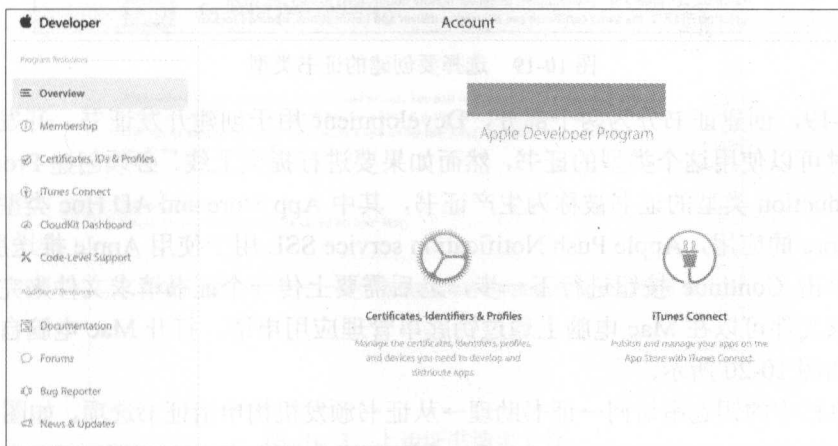


图 10-17 开发和账户管理界面

首先在证书管理界面创建一个新的证书，如图 10-18 所示。

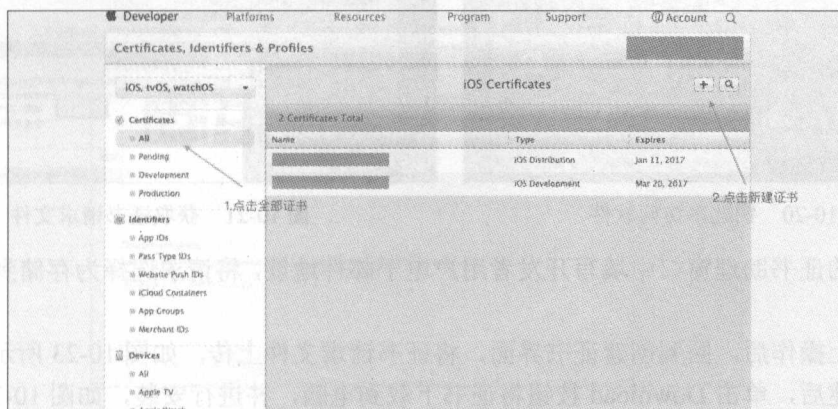


图 10-18 新建证书

之后会弹出创建证书类型选择界面，如图 10-19 所示。

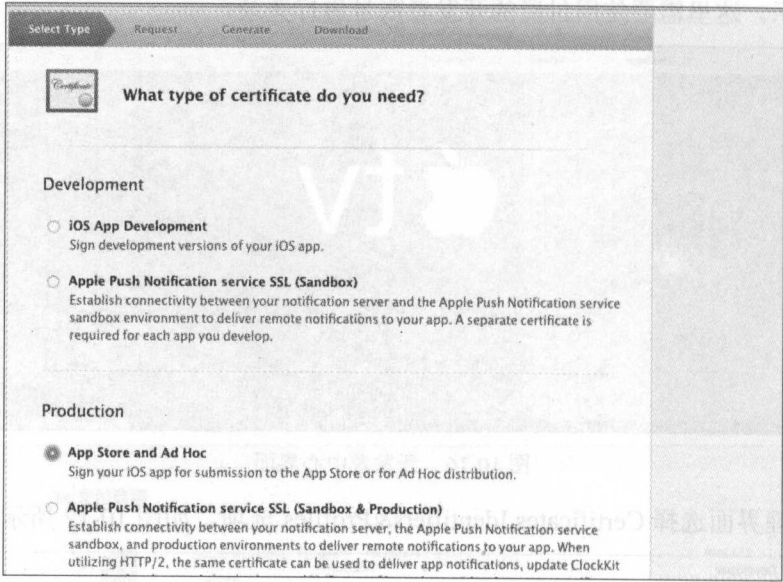


图 10-19 选择要创建的证书类型

如图 10-19，创建证书分为两个部分，Development 用于创建开发证书，开发者在开发应用进行调试时可以使用这个类型的证书，然而如果要进行提交上线，必须创建 Production 类型的证书，Production 类型的证书被称为生产证书，其中 App Store and AD Hoc 类型用于普通的提交 AppleStore 的应用，Apple Push Notification service SSL 用于使用 Apple 推送服务的应用。选择完成后单击 Continue 按钮进行下一步，之后需要上传一个证书请求文件来完成证书的创建，证书请求文件可以在 Mac 电脑上通过钥匙串管理应用申请。打开 Mac 电脑自带的软件钥匙串访问，如图 10-20 所示。

选择菜单栏中的钥匙串访问→证书助理→从证书颁发机构申请证书选项，如图 10-21 所示。

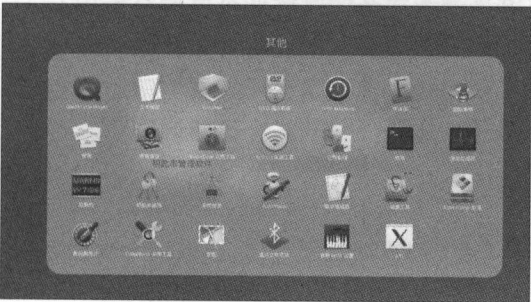


图 10-20 钥匙串访问软件



图 10-21 获取证书请求文件

在弹出的证书助理窗口中填写开发者用户电子邮件地址，将请求选择为存储到磁盘，如图 10-22 所示。

完成如上操作后，回到创建证书界面，将证书请求文件上传，如图 10-23 所示。

完成创建后，单击 Download 按钮将证书下载到电脑，并进行安装，如图 10-24 所示。

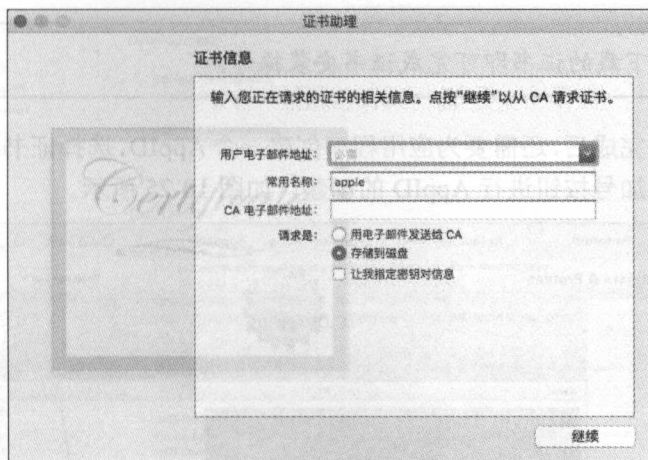


图 10-22 将证书请求文件保存到磁盘

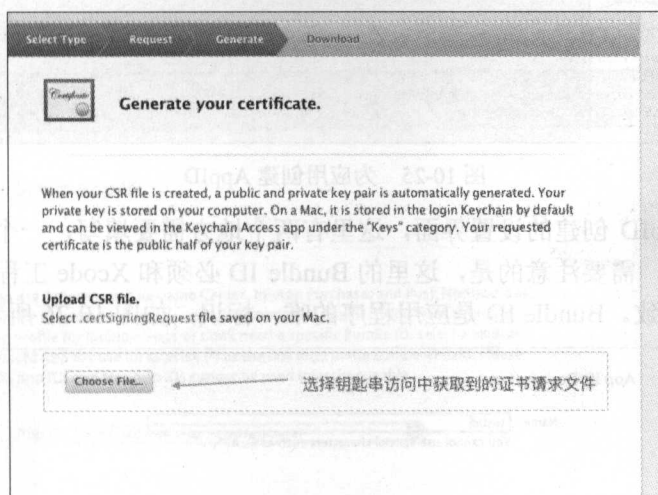


图 10-23 上传证书请求文件

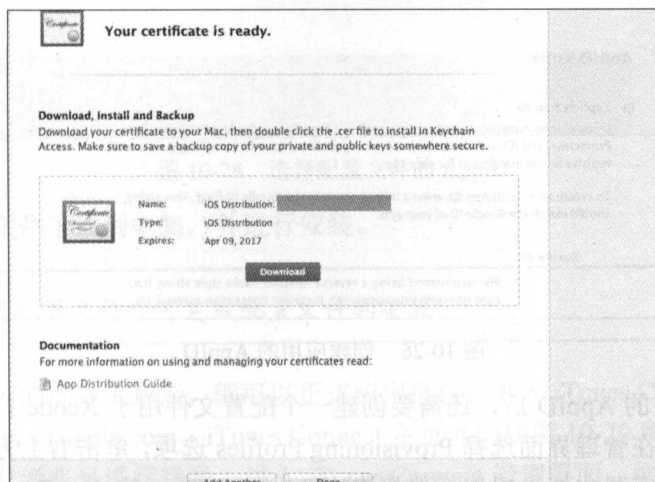


图 10-24 将证书下载至电脑



提示

双击下载的证书即可完成证书安装操作。

证书创建并设置完成后,还需要为应用程序创建一个 AppID,选择证书管理界面的 App IDs 选项,单击右上方的加号按钮进行 AppID 的创建,如图 10-25 所示。

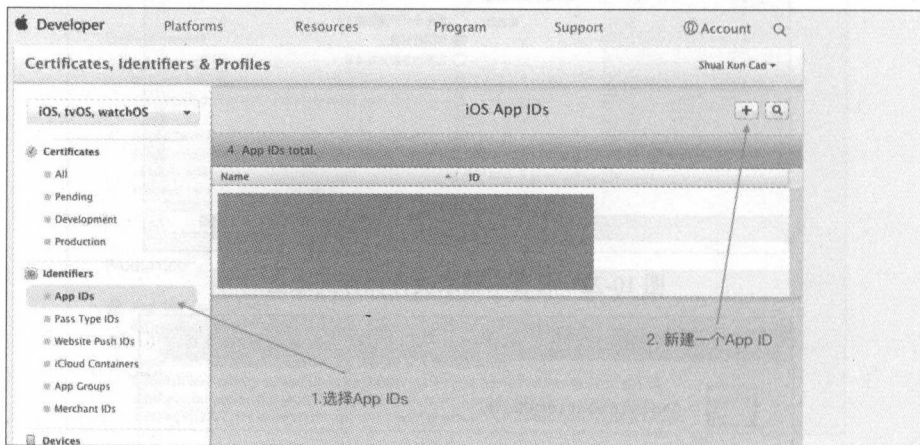


图 10-25 为应用创建 AppID

之后会弹出 AppID 创建的设置界面,这里有两个地方需要填写,一个为名称,一个为对应应用的 Bundle ID,需要注意的是,这里的 Bundle ID 必须和 Xcode 工程的 info.plist 文件中的 Bundle ID 完全一致。Bundle ID 是应用程序的唯一标识,如图 10-26 所示。

App ID Description

Name:

You cannot use special characters such as @, &, *, ',"

App ID Prefix

Value: 3637NJMX7D (Team ID)

App ID Suffix

☒ Explicit App ID

If you plan to incorporate app services such as Game Center, In-App Purchase, Data Protection, and iCloud, or want a provisioning profile unique to a single app, you must register an explicit App ID for your app.

To create an explicit App ID, enter a unique string in the Bundle ID field. This string should match the Bundle ID of your app.

Bundle ID:

We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*).

图 10-26 创建应用的 AppID

创建完应用程序的 AppID 后,还需要创建一个配置文件用于 Xcode 打包,和创建证书与 AppID 的方法类似,在管理界面选择 Provisioning Profiles 选项,单击右上方的加号按钮创建一个新的配置文件,在配置文件设置界面选择 Distribution 选项下的 App Store,如图 10-27 所示。这里 Development 下的配置文件用于开发者开发测试使用, Distribution 用于上线生产使用。

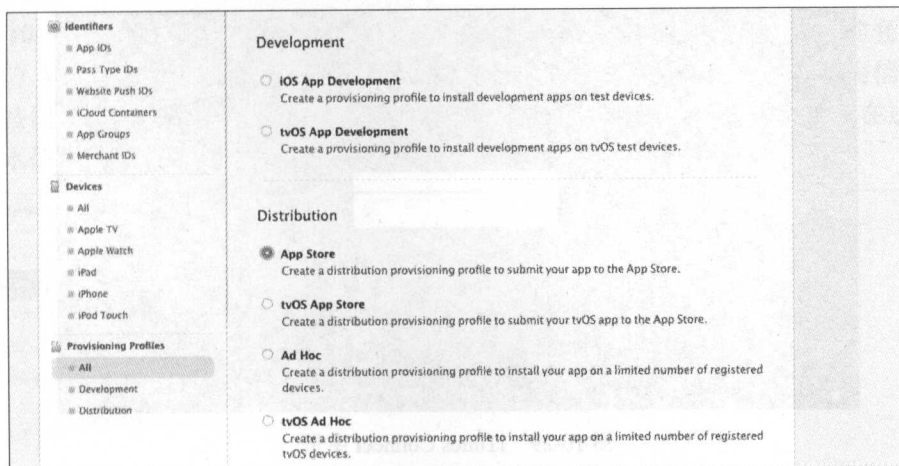


图 10-27 创建配置文件

创建配置文件时需要选择 AppID，这里要选择上面创建的 AppID，如图 10-28 所示。

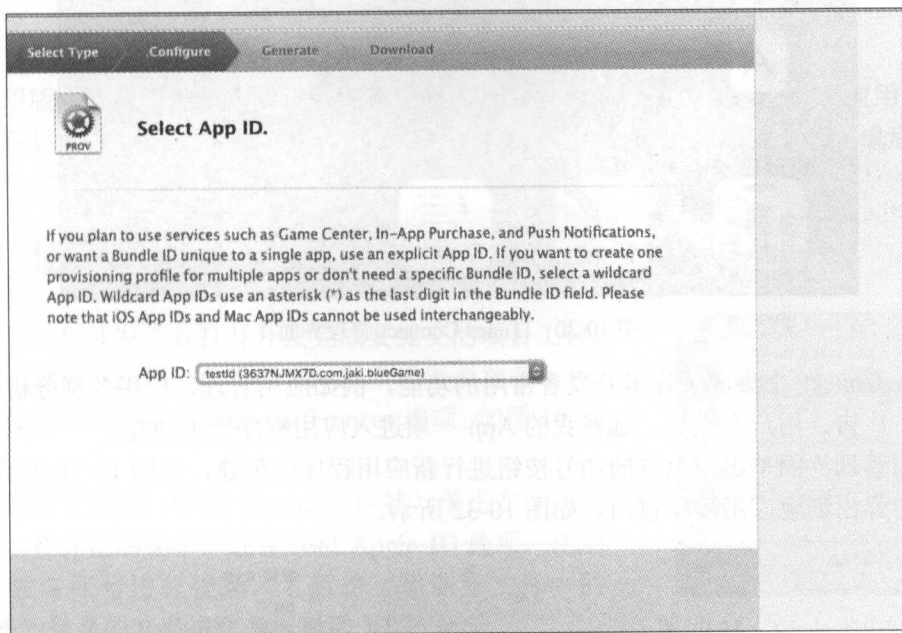


图 10-28 选择配置文件的 AppID

将创建的配置文件下载到电脑，并进行安装。



提示

双击配置文件即可完成配置文件的安装。

做完上面一系列的准备工作后，就可以正式应用创建，进入 iTunes Connect 首页，网址如下：<https://itunesconnect.apple.com>。iTunes Connect 主页界面如图 10-29 所示。

使用申请的开发者账号进行登录，进入 iTunes Connect 管理界面，如图 10-30 所示。

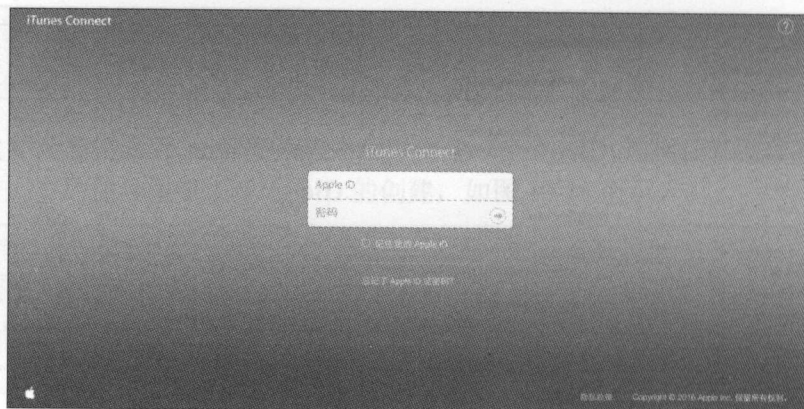


图 10-29 iTunes Connect 主页

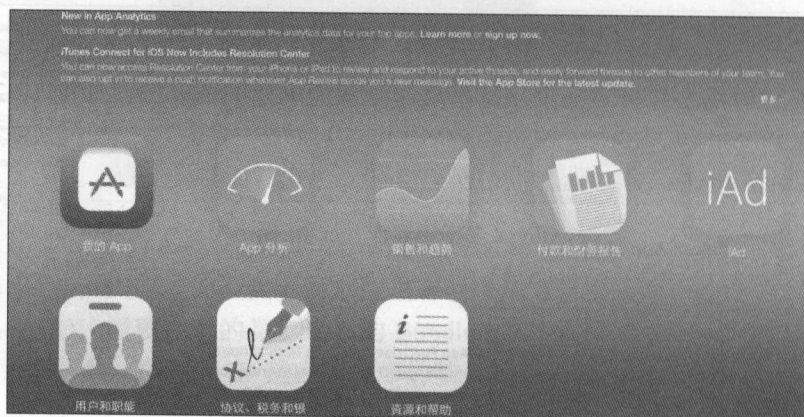


图 10-30 iTunes Connect 管理界面

iTunes Connect 中集成了许多开发者常用的功能，例如应用管理，应用数据分析与统计，收入统计，广告，用户管理等。选择我的 App 一项进入应用程序管理功能。

在应用管理界面单击左上方的加号按钮进行新应用程序的创建，如图 10-31 所示。之后会弹出创建应用程序窗口，如图 10-32 所示。



图 10-31 新建应用程序



图 10-32 新建应用创建菜单

在图 10-31 所示菜单中填写应用运行的平台、名称、语言和套装 ID，这里的套装 ID 即为前面创建的 AppID。SKU 为应用内 ID，供公司内部使用，没有具体的要求，填写信息无误后，单击创建按钮，进行应用程序的创建。单击新创建的应用，进入应用管理界面，在这里可以进行应用基本信息的设置与构建版本的提交审核，如图 10-33 所示。

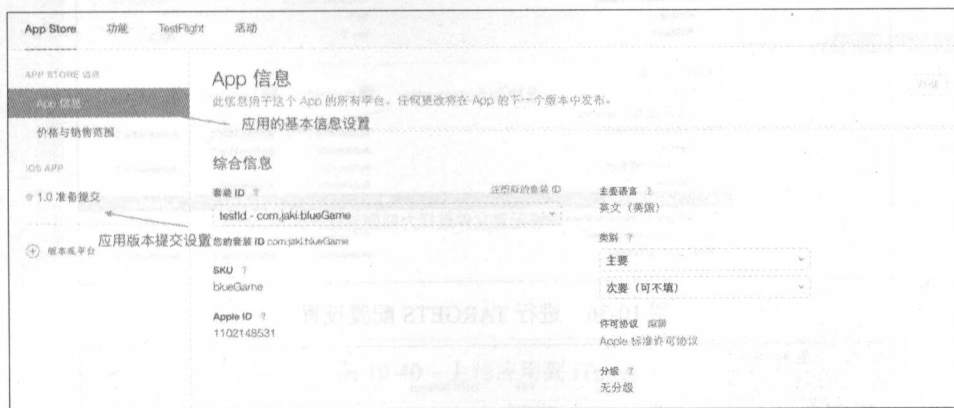


图 10-33 应用设置界面

在应用版本提交设置中需要设置应用在各个尺寸设备上的内容截图，如果应用上线成功，这些截图将显示在 AppleStore 的应用介绍界面中，将这个界面中的版本设置完成后，就可以通过 Xcode 进行打包提交操作了。

10.3.2 使用 Xcode 进行打包与提交 iTunes

使用 Xcode 开发工具打开开发完成要提交的项目工程，如果是第一次提交应用，需要在 Xcode 中设置开发者账号，单击 Xcode 菜单栏中的 Xcode-Preferences 选项，如图 10-34 所示。

选择偏好设置窗口中的 Accounts 标签，单击左下角的加号按钮，在弹出的菜单中选择 Add Apple ID 选项，使用付费的开发者账号进行登录。之后将工程配置文件中的 Team 选择为付费的开发者账号，如图 10-35 所示。

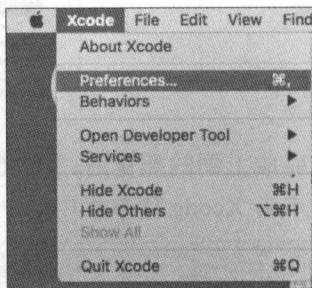


图 10-34 Xcode 的偏好设置功能

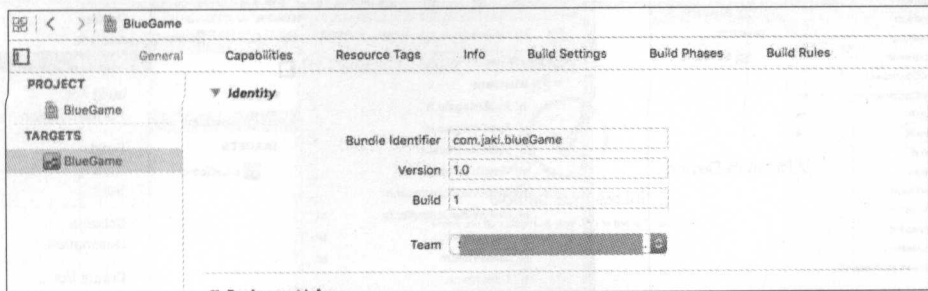


图 10-35 设置工程的 Team

将工程的打包配置文件选择为前面为此应用创建的配置文件，如图 10-36 与图 10-37 所示。

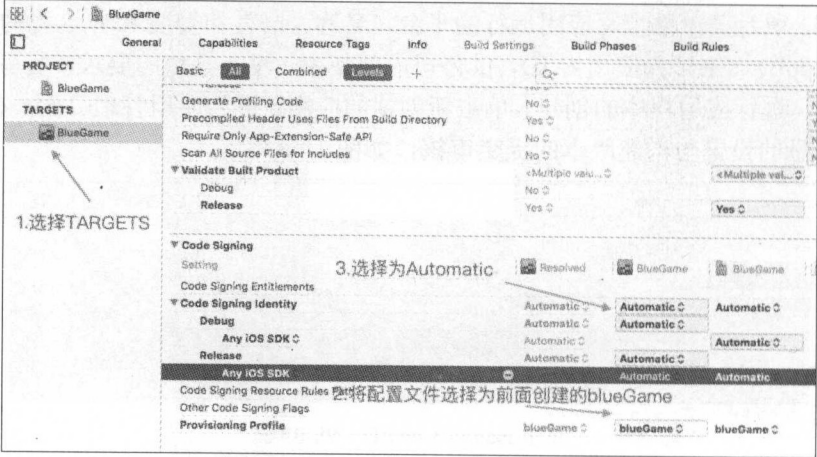


图 10-36 进行 TARGETS 配置设置



图 10-37 进行 PROJECT 配置设置

完成上面的配置后，将 Xcode 的运行设备选择为 iOS Device，如图 10-38 所示。
单击 Xcode 菜单栏中的 Product-Archive 选项，如图 10-39。



图 10-38 选择 Xcode 的运行设备

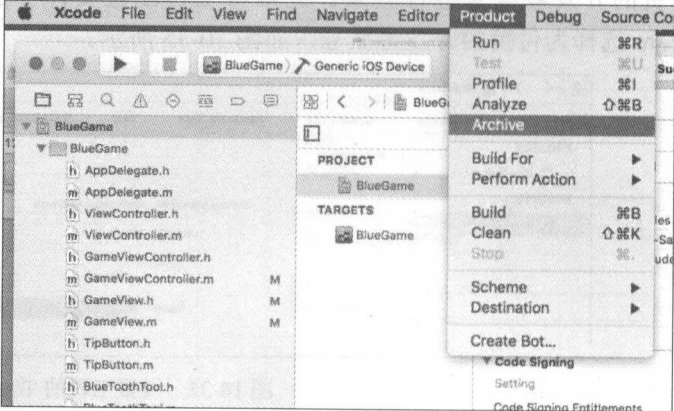


图 10-39 进行应用打包

等待一段时间,如果打包成功,会出现如图 10-40 所示的界面,单击右侧的 Upload to App Store 选项进行应用上传。

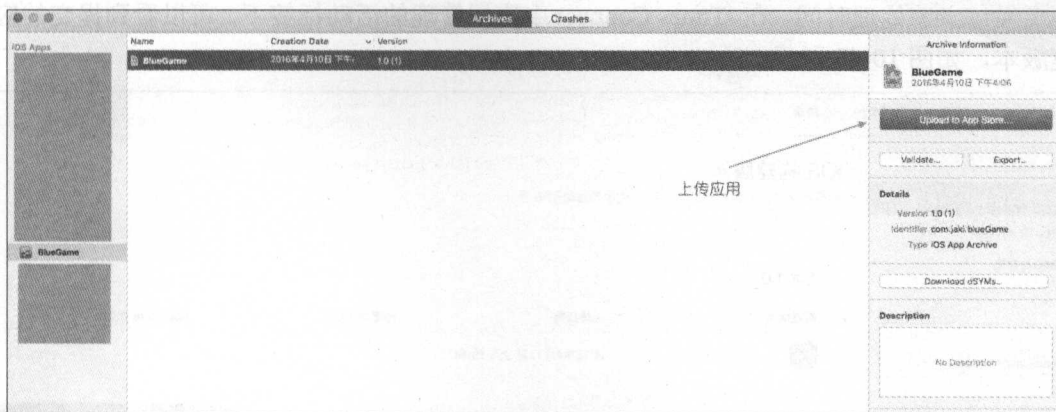


图 10-40 上传应用到 iTunes

单击 Upload to App Store 按钮后,会弹出选择开发者账号的窗口,这里也要选择与前边配置一致的付费开发者账号。之后会弹出确认上传窗口,单击 Upload 进行正式上传,如图 10-41 所示。

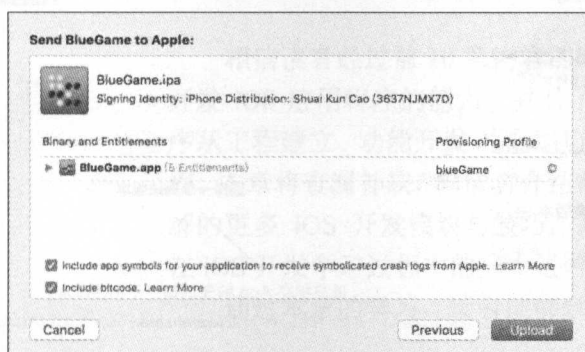


图 10-41 确认上传应用

应用的上传可能会耗费很长的时间,耐心等待,上传成功后会弹出如图 10-42 所示的成功界面。

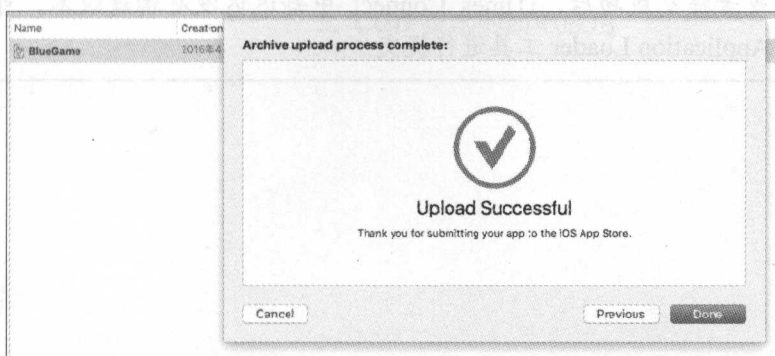


图 10-42 上传应用成功

此处的上传是将应用程序版本上传到 iTunes Connect，开发者还需要在 iTunes Connect 中进行提交审核。刚上传到 iTunes 的应用并不一定会马上出现在 iTunes 的构建版本中，在这之间需要经过一些处理时间。在 iTunes Connect 中应用管理的活动标签下，可以看到提交的应用构建版本，如图 10-43 所示。

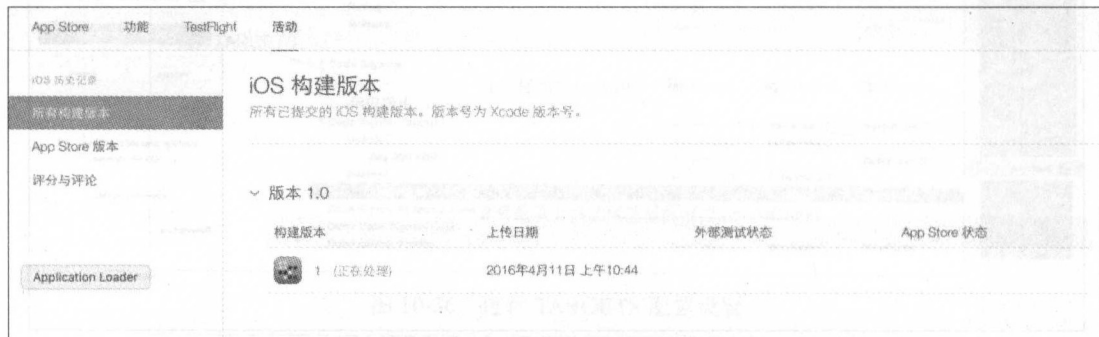


图 10-43 iTunes Connect 中提交的应用构建版本

在提交版本设置中选择一个构建版本，如图 10-44 所示。之后确认版本信息设置无误后，单击界面右上方提交以供审核按钮将应用提交审核。

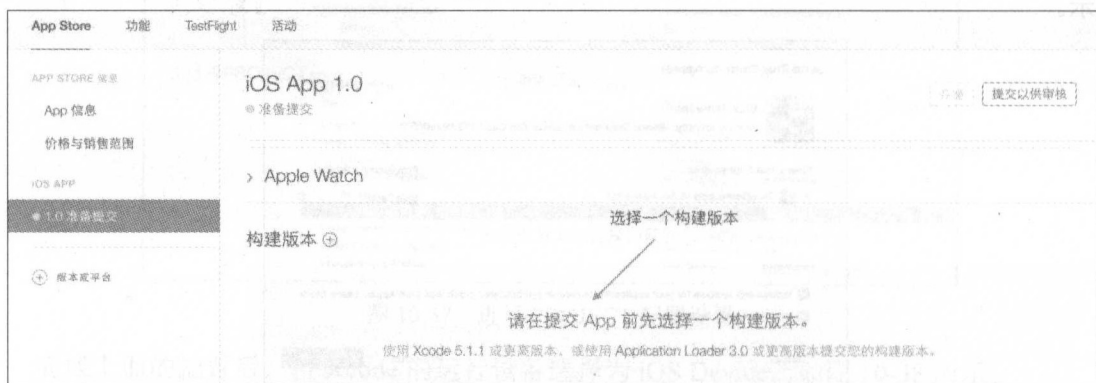


图 10-44 选择一个构建版本提交审核



提示

iTunes Connect 服务器时长会不太稳定，如果使用 Xcode 打包提交总是超时，或者提交成功后，iTunes Connect 中却迟迟没有构建版本，可以选择使用 Application Loader 工具进行提交。

第 11 章

更多功能与进阶技巧

相信读者经过前 10 章内容的学习，已经具备了独立开发 iOS 应用程序的能力，并且了解了一款 iOS 应用程序从工程建立、功能开发、测试打包到提交上线的全部过程。本章将查漏补缺，向读者介绍前面章节中没有具体讲解的更多 iOS 开发模块与技巧，同时，本章也将是读者在 iOS 开发学习过程中的一个进阶与提高。

通过本章的学习，读者能够掌握：

1. block 代码结构的应用
2. 通过通知中心进行应用值与逻辑的传递
3. 能够使用线程管理技术 NSThread 进行多线程开发
4. 能够使用任务队列 NSOperation 进行多任务开发
5. CGD 技术的简单应用

11.1 Objective-C 中 block 语法的应用

block 代码块是 Objective-C 语言的特点之一，使用 block 代码块可以十分方便地处理回调逻辑。Objective-C 语言引入 block 语法后，iOS 开发框架中很多的框架也都采用了这种设计风格。在前边介绍 UIView 层动画的章节中，读者就已经尝试过了 block 的方便之处。本节将继续向读者深入介绍 block 语法的使用与注意事项。

11.1.1 声明与实现 block 语法块

block 是一种语法，也是一种数据类型，只是 block 数据类型对应的是代码块，从这一特点来看，block 又十分类似函数指针。使用 Xcode 创建一个名为 BlockTest 的工程。声明一个 block 类型指针，使用如下的语法格式：

```
type (^name) (param1,param2,...)
```

在上面语法格式中，type 对应此 block 语法块的返回值，name 对应所声明的 block 变量的名称，param 系列对应 block 语法块的参数，可以为多个。例如，如下声明的 block 类型代表有两个 int 类型的参数，返回值为 int 类型的代码块，命名为 myBlock：

```
int (^myBlock) (int,int)
```

同样，也可以为 block 语法块中的参数取一个名字，如下所示。

```
int (^myBlock) (int a,int b);
```

在实际开发中，很多时候都需要将 block 代码块作为参数传递进某个函数中，这时需要将函数的参数设置为对应的 block 对象，示例如下所示。

```
-(void)func:(int (^)(int a,int b))block;
```

读者也可以发现，直接将不加修饰的 block 变量作为函数的参数这种写法可读性并不好，因此，开发者常常会使用 typedef 来定义一个 block 类型，在将其作为参数传递进函数，这样可以使代码更加美观整洁，示例如下所示。

```
typedef int (^myBlock) (int a,int b) ;
-(void)func:(myBlock)block ;
```

block 既然可以被声明为变量，那它也应该向其他常规变量一样，可以被赋值，被修改。与常规变量不同的是，block 的赋值实际上赋的是一个代码块，因此，对 block 变量的赋值也可以叫做对 block 变量的实现。

在前边创建的 BlockTest 工程的 ViewController.m 文件中声明一个 block 变量如下所示。

```
typedef int (^myBlock) (int,int) ;
@interface ViewController ()
```

```

{
    myBlock block1;
}
@end

```

在 `viewDidLoad` 方法中对 `block1` 变量进行如下赋值。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    block1 = ^(int a, int b){
        return a+b;
    };
    NSLog(@"%d",block1(1,1));
}

```

`NSLog` 打印的数据即为 `block` 代码块的返回值。



提示

`block1` 是一个 `block` 对象，如果直接打印 `block1`，将会打印出对象的地址。
`block()` 才是执行 `block` 代码块，加上括号才能打印出返回值。

11.1.2 block 代码块中访问对象的微妙关系

在 `block` 代码块中经常需要访问到 `block` 外的对象与数据，如果在 `block` 中只是对外部对象的读取访问，是没有任何问题的，如下所示。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    int tmp=2;
    block1 = ^(int a, int b){
        return a+b+tmp;
    };
    NSLog(@"%d",block1(1,1));
}

```

如果在 `block` 中进行外部变量的修改，读者就会发现一个微妙的问题，例如编写如下代码，Xcode 将会报错。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    int tmp=2;
    block1 = ^(int a, int b){
        tmp = 3;
        return a+b+tmp;
    };
    NSLog(@"%d",block1(1,1));
}

```

Xcode 报错的原因是外部变量不能在 block 内被修改。事实上，当一个外部变量被 block 访问时，为防止在执行 block 代码块时外部的变量被释放，进入 block 的变量会被系统自动的复制一份，因此，在 block 内的变量实际上已经不是外部变量本身，而是外部变量的一个副本，修改这个副本将达不到开发者预期的效果，并且 Xcode 开发工具也会报出错误。如果需要在 block 内部对外部变量做修改操作，需要将外部变量修饰为 `__block` 类型，如下所示。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    __block int tmp=2;
    block1 = ^(int a, int b){
        tmp = 3;
        return a+b+tmp;
    };
    NSLog(@"%d",block1(1,1));
}
```

上面代码的运行将不再有任何问题。

如果 block 需要使用的外部变量为 Objective-C 可变对象，则情况又有不同，例如下面代码运行也是没有问题的。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSMutableString * str = [[NSMutableString alloc] initWithString:@"one"];
    block1 = ^(int a, int b){
        [str appendString:@"two"];
        NSLog(str);
        return 0;
    };
    NSLog(@"%d",block1(1,1));
}
```

上面代码虽然对可变字符串对象 `str` 做了修改，但是对象本身没有变，指针 `str` 地址并没有更改，因此代码运行时没有问题的。

简单来说，`__block` 修饰符做了这样一件事：如果一个对象没有声明为 `__block` 类型，则 block 对它访问时会复制一份进行访问，如果一个对象声明为了 `__block` 类型，则在 block 对其进行访问时，是会访问其地址对应的数据。

11.2 iOS 通知中心 NSNotificationCenter 的应用

通知是 iOS 开发中一种重要的传值手段，并且许多复杂的多界面交互逻辑，使用通知都可以十分方便地进行处理。

11.2.1 通知类 NSNotification 简介

NSNotification 类可以理解为一个通知对象，通知也是一种消息。对于通知其实读者并不陌生，iOS 系统中本就注册了许多通知，例如当键盘弹出或者收起时系统会发送相应通知，当物体靠近和远离距离传感器时，系统也会发送相应的通知。开发者在进行应用开发时，除了可以直接监听系统注册的通知之外，也可以自己新建通知，在需要的时候使用通知中心进行通知的发送。

使用 Xcode 创建一个名为 NSNotificationTest 的工程。一个 NSNotification 对象必须有一个名字 name，一个可选的用户字典 userInfo 与一个可选的针对对象 object。在 NSNotificationTest 工程的 ViewController.m 文件的 viewDidLoad 方法中创建两个通知对象，实例代码如下所示。

```
//创建无用户字典的通知对象
NSNotification * noti = [NSNotification notificationWithName:@"notification" object:nil];
//创建有用户字典的通知对象
NSNotification * noti2 = [NSNotification notificationWithName:@"notification2" object:nil userInfo:@{@"key":@"value"}];
```

上面两个方法分别创建了不带用户字典的 NSNotification 对象与带 NSNotification 用户字典的 NSNotification 对象。通知对象的 name 参数用于标识通知本身，userInfo 参数可以让通知在发送过程中传递一些数据，通知的监听者可以获取到通知的用户字典进行逻辑处理。

除了上面两个创建通知对象的类方法外，也可以通过如下实例方法来初始化通知对象。

```
NSNotification * noti3 = [[NSNotification alloc] initWithName:@"notification3" object:nil userInfo:@{@"key":@"value"}];
```

11.2.2 通知中心 NSNotificationCenter 应用

在上一小节中，向读者介绍了通知对象 NSNotification 的创建，NSNotification 对象从来都是与 NSNotificationCenter 进行结合使用的，NSNotificationCenter 通知中心负责整体协调与发送通知对象。NSNotificationCenter 类采用单例的设计模式，每个应用程序都有一个默认的 NSNotificationCenter 通知中心对象，其作用于是整个应用程序。

将 NSNotificationTest 工程的 viewDidLoad 方法中的代码修改如下所示。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    //为一个特定的通知添加监听者
    [[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(noti:) name:@"noti" object:nil];
}
```

NSNotificationCenter 类的 defaultCenter 用于获取单例对象。addObserver:selector:name:object: 方法用于为某个通知添加一个监听者，这个方法中的第 1 个参数设置监听者对象，即执行监听

方法的类对象，第 2 个参数为收到通知后执行的方法选择器，第 3 个参数为所监听的通知的名称。第 4 个参数需要与通知对象的 `object` 参数对应。实现接收到通知后出发的方法 `noti` 如下所示。

```
-(void)noti:(NSNotification *)noti{
    NSLog(@"%@",noti.userInfo);
}
```

为了测试通知的发送与接收，在 `ViewController.m` 文件中实现 `touches:beganWithEvent:` 方法如下所示。

```
-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event{
    NSNotification * noti = [NSNotification notificationWithName:@"noti" o
bject:nil userInfo:@{@"message":@"HelloWorld"}];
    //发送通知
    [[NSNotificationCenter defaultCenter]postNotification:noti];
}
```

运行程序，单击屏幕，可以看到 Xcode 调试区打印出的用户字典信息。上面方法使用 `NSNotificationCenter` 单例对象调用 `postNotification:` 方法之间发送一个 `NSNotification` 通知对象，`NSNotificationCenter` 中还提供了如下两种方法进行通知的发送。

```
-(void)postNotificationName:(NSString *)aName object:(id)anObject;
-(void)postNotificationName:(NSString *)aName object:(id)anObject userInfo:(NSDictionary *)aUserInfo;
```

当某个监听者不需要再监听某个通知时，可以使用如下方法来移除通知的监听者。

```
-(void)removeObserver:(id)observer;
-(void)removeObserver:(id)observer name:(NSString *)aName object:(id)anObject;
```

上面列举出的两个方法中，第 1 个方法移除通知中心的某个监听者，这个监听者将不再监听所有通知，第 2 个方法将移除某个监听者对某个特定通知的监听，并不影响其监听其他通知。例如，监听者 X 监听了 A 和 B 两个通知，可以使用第 2 个方法只将对 A 通知的监听移除。

11.3 多线程开发技术

在应用开发中，开发者若想更加深入地理解与控制程序的运行机理，就不得不了解与学习进程与线程的相关内容。在 iOS 系统中，一个运行着的应用程序就可以理解为一个程序进程，在同一时间，iOS 系统只允许一个进程处于激活状态，当然，在 iOS 9 之后，系统支持了类似画中画技术的多进程处理方案。线程是进程中的概念，一个进程在执行任务时，可以通知并行开启多个线程，一个线程就是一条任务线。

11.3.1 使用 NSThread 进行线程管理

NSThread 是 iOS 开发框架中层次较低的一个线程管理类，开发者在使用其进行线程管理的时候需要自行关注线程安全，生命周期等问题，并且其性能表现也并不十分优秀。后面章节会向读者介绍 iOS 开发中一种更加流行也更加强大的多线程开发技术——GCD。

多线程开发的一个重要用途是在程序中并行的执行多任务，例如大部分下载请求任务都是在单独的线程中执行的，这样不会阻塞 UI 界面的渲染。同样，一些计算量大的耗时任务，通常也会放在单独的线程中执行。使用 Xcode 创建一个名为 NSThreadTest 的工程。在 ViewController.m 文件的 viewDidLoad 方法中编写如下测试代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    [NSThread detachNewThreadSelector:@selector(newThread) toTarget:self
    withObject:nil];
    for (int i=0; i<10; i++) {
        NSLog(@"主线程: %@ %d", [NSThread currentThread], i);
    }
}
```

选择器方法 newThread 方法的实现如下所示。

```
-(void)newThread{
    for (int i=0; i<10; i++) {
        NSLog(@"子线程: %@ %d", [NSThread currentThread], i);
    }
    //任务结束后要结束线程
    [NSThread exit];
}
```

在上面的演示代码中，如果抛开线程的影响，代码的执行顺序应该先执行 newThread 方法中的 10 次循环打印，再执行 viewDidLoad 方法中的 10 次循环打印。然而实际上，newThread 方法是在一个独立的子线程中执行的，因此两次循环打印应该同时进行执行。上面代码中，NSThread 类的类方法 detachNewThreadSelector:toTarget:withObject:方法用于创建一个匿名线程来执行传入的选择器方法。在 newThread 方法中，任务执行完毕后，调用 NSThread 类的 exit 方法来结束当前线程。

运行工程，可以看到在 Xcode 调试区打印出如图 11-1 所示的信息，从打印信息中可以看出，两部分循环是同时执行的。

NSThread 类的 currentThread 方法可以获取到执行当前任务的线程。也可以调用如下方法使当前线程休眠一段时间再执行。

```
+ (void)sleepUntilDate:(NSDate *)date;
+ (void)sleepForTimeInterval:(NSTimeInterval)ti;
```



```

2016-04-14 16:27:54.436 NSThreadTest[7230:176042] 主线程: <NSThread: 0x7fc552604fa0>{number = 1, name = main} 0
2016-04-14 16:27:54.436 NSThreadTest[7230:176099] 子线程: <NSThread: 0x7fc552517670>{number = 2, name = (null)} 0
2016-04-14 16:27:54.436 NSThreadTest[7230:176042] 主线程: <NSThread: 0x7fc552604fa0>{number = 1, name = main} 1
2016-04-14 16:27:54.436 NSThreadTest[7230:176099] 子线程: <NSThread: 0x7fc552517670>{number = 2, name = (null)} 1
2016-04-14 16:27:54.436 NSThreadTest[7230:176042] 主线程: <NSThread: 0x7fc552604fa0>{number = 1, name = main} 2
2016-04-14 16:27:54.436 NSThreadTest[7230:176099] 子线程: <NSThread: 0x7fc552517670>{number = 2, name = (null)} 2
2016-04-14 16:27:54.437 NSThreadTest[7230:176042] 主线程: <NSThread: 0x7fc552604fa0>{number = 1, name = main} 3
2016-04-14 16:27:54.437 NSThreadTest[7230:176099] 子线程: <NSThread: 0x7fc552517670>{number = 2, name = (null)} 3
2016-04-14 16:27:54.437 NSThreadTest[7230:176042] 主线程: <NSThread: 0x7fc552604fa0>{number = 1, name = main} 4
2016-04-14 16:27:54.437 NSThreadTest[7230:176099] 子线程: <NSThread: 0x7fc552517670>{number = 2, name = (null)} 4
2016-04-14 16:27:54.437 NSThreadTest[7230:176042] 主线程: <NSThread: 0x7fc552604fa0>{number = 1, name = main} 5
2016-04-14 16:27:54.437 NSThreadTest[7230:176099] 子线程: <NSThread: 0x7fc552517670>{number = 2, name = (null)} 5
2016-04-14 16:27:54.506 NSThreadTest[7230:176042] 主线程: <NSThread: 0x7fc552604fa0>{number = 1, name = main} 6

```

图 11-1 主线程与子线程同时进行任务执行

`sleepUntilData`:方法设置当前线程休眠到某一时间再执行任务, `sleepForTimeInterval`:方法设置线程休眠一定时间再执行任务。

除了使用上面示例的类方法来创建匿名线程执行任务外,也可以使用如下初始化方法来创建一个新的 `NSThread` 线程对象:

```

- (instancetype)initWithTarget:(id)target selector:(SEL)selector object:
(id)argument;

```

上面所提及的 `NSThread` 方法都是显式的调用 `NSThread` 类的相关方法进行创建线程与执行任务,其实 `NSObject` 类的扩展中定义好了一些方法,开发者可以调用这些方法进行多线程执行任务,因为使用这种方式并没有明显的调用 `NSThread` 类的相关方法,这种方式也被称为隐式 `NSThread` 编程。常用方法示例如下所示。

```

- (void)performSelectorOnMainThread:(SEL)aSelector withObject:(id)arg wait
UntilDone:(BOOL)wait;

```

上面方法在主线程中执行选择器传入的方法任务, `wait` 参数为是否等待主线程任务完成,如果设置为 `YES`,则等待主线程中正在进行的任务结束后再执行,设置为 `NO` 则立即执行。

```

- (void)performSelector:(SEL)aSelector onThread:(NSThread *)thr withObject:
t:(id)arg waitUntilDone:(BOOL)wait;

```

上面的方法将指定在一个特定的线程中执行选择器传入的方法。

```

- (void)performSelectorInBackground:(SEL)aSelector withObject:(id)arg;

```

上面方法将在后台线程中执行选择器传入的方法。

11.3.2 使用 `NSOperation` 类与 `NSOperationQueue` 类进行多任务管理

使用 `NSOperation` 类与 `NSOperationQueue` 类也是 iOS 中多线程编程的一种手段, `NSOperation` 和 `NSOperationQueue` 是基于 Objective-C 风格的一套管理与执行任务的框架。 `NSOperation` 类是一个抽象类,通常开发者会使用 `NSInvocationOperation` 与 `NSBlockOperation` 这两个子类进行多任务开发,当然,开发者也可以编写继承于 `NSOperation` 的自定义子类来实现逻辑操作。

NSInvocationOperation 执行的操作就在当前线程中执行,与当前线程的其他任务是同步执行的。使用 Xcode 创建一个名为 NSInvocationOperationTest 的工程,在 ViewController.m 文件的 viewDidLoad 方法中编写如下测试代码:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    //创建与初始化 NSInvocationOperation 对象
    NSInvocationOperation * operation = [[NSInvocationOperation alloc] initWithTarget:self selector:@selector(task) object:nil];
    //开始执行任务
    [operation start];
    for (int i=0; i<10; i++) {
        NSLog(@"%d", [NSThread currentThread], i);
    }
}
```

上面代码中,调用 NSInvocationOperation 类对象的 start 方法用于开始执行任务,实现 task 方法如下所示。

```
-(void)task{
    for (int i=0; i<10; i++) {
        NSLog(@"%d", [NSThread currentThread], i);
    }
}
```

在 Xcode 的调试区可以看到如图 11-2 的打印信息。

```
2016-04-16 11:57:16.712 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=0
2016-04-16 11:57:16.713 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=1
2016-04-16 11:57:16.713 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=2
2016-04-16 11:57:16.713 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=3
2016-04-16 11:57:16.713 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=4
2016-04-16 11:57:16.713 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=5
2016-04-16 11:57:16.713 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=6
2016-04-16 11:57:16.713 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=7
2016-04-16 11:57:16.713 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=8
2016-04-16 11:57:16.713 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=9
2016-04-16 11:57:16.770 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=0
2016-04-16 11:57:16.770 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=1
2016-04-16 11:57:16.770 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=2
2016-04-16 11:57:16.770 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=3
2016-04-16 11:57:16.771 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=4
2016-04-16 11:57:16.771 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=5
2016-04-16 11:57:16.771 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=6
2016-04-16 11:57:16.771 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=7
2016-04-16 11:57:16.771 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=8
2016-04-16 11:57:16.771 NSInvocationOperationTest[808:57762] <NSThread: 0x7fd780d08060>{number = 1, name = main}=9
```

图 11-2 Xcode 的打印信息

从打印信息中可以看出,task 任务与后面的 for 循环都是在主线程中执行的,并且是同步执行的,当 task 任务结束后,才开始继续执行后面的 for 循环。



提示 同步指执行的任务一个结束后再执行另外一个,异步是指几项任务同时执行。

NSBlockOperation 类相比 NSInvocationOperation 类处理多任务的能力要强大的多,NSBlockOperation 可以在执行前向其中添加多个 block 任务块。使用 Xcode 创建一个名为

NSBlockOperationTest 的工程，在 ViewController.m 文件的 viewDidLoad 方法中添加如下所示的测试代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    NSBlockOperation * operation = [NSBlockOperation blockOperationWithBlock:^(
        for (int i=0; i<10; i++) {
            NSLog(@"1:%@=%d", [NSThread currentThread], i);
        }
    )];
    [operation addExecutionBlock:^(
        for (int i=0; i<10; i++) {
            NSLog(@"2:%@=%d", [NSThread currentThread], i);
        }
    )];
    [operation addExecutionBlock:^(
        for (int i=0; i<10; i++) {
            NSLog(@"3:%@=%d", [NSThread currentThread], i);
        }
    )];
    [operation start];
}
```

在上面的代码中，除了在创建 NSBlockOperation 时初始化了一个 block 任务外，后面又使用 addExecutionBlock: 方法追加了两个 block 任务，运行工程，打印信息如图 11-3 所示。

```
2016-04-16 12:22:39.485 NSBlockOperationTest[928:70824] 1:<NSThread: 0x7fed4a900640>{number = 1, name = main}=0
2016-04-16 12:22:39.485 NSBlockOperationTest[928:70865] 2:<NSThread: 0x7fed4a806150>{number = 3, name = (null)}=0
2016-04-16 12:22:39.485 NSBlockOperationTest[928:70856] 3:<NSThread: 0x7fed48c192e0>{number = 2, name = (null)}=0
2016-04-16 12:22:39.485 NSBlockOperationTest[928:70824] 1:<NSThread: 0x7fed4a900640>{number = 1, name = main}=1
2016-04-16 12:22:39.485 NSBlockOperationTest[928:70865] 2:<NSThread: 0x7fed4a806150>{number = 3, name = (null)}=1
2016-04-16 12:22:39.485 NSBlockOperationTest[928:70856] 3:<NSThread: 0x7fed48c192e0>{number = 2, name = (null)}=1
2016-04-16 12:22:39.486 NSBlockOperationTest[928:70824] 1:<NSThread: 0x7fed4a900640>{number = 1, name = main}=2
2016-04-16 12:22:39.486 NSBlockOperationTest[928:70865] 2:<NSThread: 0x7fed4a806150>{number = 3, name = (null)}=2
2016-04-16 12:22:39.486 NSBlockOperationTest[928:70856] 3:<NSThread: 0x7fed48c192e0>{number = 2, name = (null)}=2
2016-04-16 12:22:39.486 NSBlockOperationTest[928:70824] 1:<NSThread: 0x7fed4a900640>{number = 1, name = main}=3
2016-04-16 12:22:39.486 NSBlockOperationTest[928:70865] 2:<NSThread: 0x7fed4a806150>{number = 3, name = (null)}=3
2016-04-16 12:22:39.486 NSBlockOperationTest[928:70856] 3:<NSThread: 0x7fed48c192e0>{number = 2, name = (null)}=3
2016-04-16 12:22:39.486 NSBlockOperationTest[928:70824] 1:<NSThread: 0x7fed4a900640>{number = 1, name = main}=4
2016-04-16 12:22:39.486 NSBlockOperationTest[928:70865] 2:<NSThread: 0x7fed4a806150>{number = 3, name = (null)}=4
2016-04-16 12:22:39.486 NSBlockOperationTest[928:70856] 3:<NSThread: 0x7fed48c192e0>{number = 2, name = (null)}=4
2016-04-16 12:22:39.486 NSBlockOperationTest[928:70824] 1:<NSThread: 0x7fed4a900640>{number = 1, name = main}=5
2016-04-16 12:22:39.486 NSBlockOperationTest[928:70865] 2:<NSThread: 0x7fed4a806150>{number = 3, name = (null)}=5
2016-04-16 12:22:39.486 NSBlockOperationTest[928:70856] 3:<NSThread: 0x7fed48c192e0>{number = 2, name = (null)}=5
2016-04-16 12:22:39.486 NSBlockOperationTest[928:70824] 1:<NSThread: 0x7fed4a900640>{number = 1, name = main}=6
```

图 11-3 Xcode 的打印信息

从打印信息中可以印证，只有初始化 NSBlockOperation 时创建的 block 任务在主线程中执行，其后再加入的 block 任务都会在独立的新线程中执行，这些任务都是异步执行的，任务完成的先后顺序并无必然关系。

除了 start 方法用于开始执行任务，NSOperation 中还有许多进行任务控制的方法，常用方法列举如下所示。

```
//取消任务执行
- (void)cancel;
//等待线程中任务结束
```



```

- (void)waitUntilFinished;
//当前操作是否取消执行
@property (readonly, getter=isCancelled) BOOL cancelled;
//当前操作是否正在执行
@property (readonly, getter=isExecuting) BOOL executing;
//当前操作是否执行结束
@property (readonly, getter=isFinished) BOOL finished;
//当前操作是否在异步线程中
@property (readonly, getter=isAsynchronous) BOOL asynchronous;
//当前操作是否已经准备好
@property (readonly, getter=isReady) BOOL ready;
//设置名称
@property (copy) NSString *name;

```

NSOperation 可以理解为任务类，NSOperationQueue 则是任务队列，这个任务队列主要负责协调执行加入其中的任务。默认添加进 NSOperationQueue 任务队列的任务都将在独立的线程中执行。使用 Xcode 创建一个名为 NSOperationQueueTest 的工程，在 ViewController.m 文件的 viewDidLoad 方法中编写如下测试代码。

```

- (void)viewDidLoad {
    [super viewDidLoad];
    NSOperationQueue * queue = [[NSOperationQueue alloc] init];
    NSInvocationOperation * op1 = [[NSInvocationOperation alloc] initWithTa
    rget:self selector:@selector(task) object:nil];
    [queue addOperation:op1];
    for (int i=0; i<10; i++) {
        NSLog(@"%@=%d", [NSThread currentThread], i);
    }
}

```

实现 task 方法如下所示。

```

- (void)task{
    for (int i=0; i<10; i++) {
        NSLog(@"%@=%d", [NSThread currentThread], i);
    }
}

```

运行工程，观察 Xcode 打印信息可以发现，加入 NSOperationQueue 中的任务在独立的线程中执行。

NSOperationQueue 类中常用方法列举如下所示。

```

//向任务队列中添加一个 NSOperation 任务对象
- (void)addOperation:(NSOperation *)op;
//向任务队列中加入一组 NSOperation 任务对象
- (void)addOperations:(NSArray *)ops waitUntilFinished:(BOOL)wait;

```

```

//向任务队列中添加一个block 任务
-(void)addOperationWithBlock:(void (^)(void))block;
//取消任务队列中所有的任务
- (void)cancelAllOperations;
//获取主线程队列
+ (NSOperationQueue *)mainQueue;
//获取当前线程队列
+ (NSOperationQueue *)currentQueue;
//设置队列名称
@property (copy) NSString *name;
//获取队列中的所有 NSOperation 任务对象
@property (readonly, copy) NSArray *operations;
//获取队列中任务个数
@property (readonly) NSUInteger operationCount;
//设置任务队列最多同时执行的任务数
@property NSInteger maxConcurrentOperationCount;
//设置暂停任务队列的执行
@property (getter=isSuspended) BOOL suspended;

```

11.3.3 iOS 中 GCD 编程技术简介

GCD 全称是 Grand Central Dispatch(伟大的中心调度)编程框架,它是一套基于 C 语言的多线程管理框架,相比于 NSThread 和 NSOperation 更加高效与强大。并且 GCD 可以很好地支持处理器的多核运算,在线程管理内部实现上更加合理。Apple 官方也更加推荐开发者使用 GCD 进行多线程编程。

GCD 机制中与 NSOperationQueue 有一些相像的地方,都是采用任务调度队列来进行多线程任务执行。开发者使用如下函数可以获取到系统主线程调度队列。

```
dispatch_queue_t dispatch_get_main_queue(void);
```

dispatch_queue_t 是 C 语言风格的线程掉队队列对象。

除了主线程调度队列之外,每一个应用程序都默认创建了几个优先级不同的全局子线程调度队列,使用如下方法获取这些子线程。

```
dispatch_queue_t dispatch_get_global_queue(long identifier, unsigned long flags);
```

上面 C 函数中第 1 个参数用于传入一个队列 ID,用于获取相应的子线程队列,第 2 个参数是留给以后扩展所用,开发和在使用时第 2 个参数传 0 即可。第 1 个参数可以选择的 ID 宏定义如下所示。

```

//优先级最高的全局队列
#define DISPATCH_QUEUE_PRIORITY_HIGH 2
//优先级中等的全局队列
#define DISPATCH_QUEUE_PRIORITY_DEFAULT 0

```

```
// 优先级低的全局队列
#define DISPATCH_QUEUE_PRIORITY_LOW (-2)
// 后台的全局队列 优先级最低
#define DISPATCH_QUEUE_PRIORITY_BACKGROUND INT16_MIN
```

一般情况下,开发者使用主线程调度队列与 4 个优先级不同的子线程调用队列完成多线程开发的需求已经足够用了,不过 GCD 中也提供了方法供开发者创建自定义的线程调度队列,使用如下方法。

```
dispatch_queue_t dispatch_queue_create(const char *label, dispatch_queue_attr_t attr);
```

在上面的方法中,第 1 个参数设置调度队列的名称,第 2 个参数设置调度队列的类型,可选设置的类型如下所示。

```
// 创建串行的调度队列
DISPATCH_QUEUE_SERIAL
// 创建并行的调度队列
DISPATCH_QUEUE_CONCURRENT
```



提示

串行队列中的任务会一个一个依次被执行,并行队列中的任务会同时被执行。

使用 Xcode 创建一个名为 GCDTest 的工程,在 ViewController.m 文件的 viewDidLoad 方法中添加如下测试代码。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    dispatch_queue_t queue = dispatch_queue_create("myQueue", DISPATCH_QUEUE_SERIAL);
    dispatch_sync(queue, ^{
        for (int i=0; i<10; i++) {
            NSLog(@"%d:1=%d", [NSThread currentThread], i);
        }
    });
    NSLog(@"1 任务");
    dispatch_async(queue, ^{
        for (int i=0; i<10; i++) {
            NSLog(@"%d:2=%d", [NSThread currentThread], i);
        }
    });
    NSLog(@"2 任务");
}
```


在上面的测试代码中，`dispatch_sync()`方法用于向调度队列中加入同步任务，即此任务是在当前线程中执行的，`dispatch_async()`方法用于向调度队列中加入异步任务，即此任务是在新的线程中执行的。运行工程，Xcode 调试区打印信息如图 11-4 所示。

```

2016-04-17 11:09:47.027 GCDTest[624:23752] <NSThread: 0x7fee6ad057e0>{number = 1, name = main}:1=0
2016-04-17 11:09:47.028 GCDTest[624:23752] <NSThread: 0x7fee6ad057e0>{number = 1, name = main}:1=1
2016-04-17 11:09:47.028 GCDTest[624:23752] <NSThread: 0x7fee6ad057e0>{number = 1, name = main}:1=2
2016-04-17 11:09:47.028 GCDTest[624:23752] <NSThread: 0x7fee6ad057e0>{number = 1, name = main}:1=3
2016-04-17 11:09:47.028 GCDTest[624:23752] <NSThread: 0x7fee6ad057e0>{number = 1, name = main}:1=4
2016-04-17 11:09:47.028 GCDTest[624:23752] <NSThread: 0x7fee6ad057e0>{number = 1, name = main}:1=5
2016-04-17 11:09:47.028 GCDTest[624:23752] <NSThread: 0x7fee6ad057e0>{number = 1, name = main}:1=6
2016-04-17 11:09:47.029 GCDTest[624:23752] <NSThread: 0x7fee6ad057e0>{number = 1, name = main}:1=7
2016-04-17 11:09:47.029 GCDTest[624:23752] <NSThread: 0x7fee6ad057e0>{number = 1, name = main}:1=8
2016-04-17 11:09:47.029 GCDTest[624:23752] <NSThread: 0x7fee6ad057e0>{number = 1, name = main}:1=9
2016-04-17 11:09:47.029 GCDTest[624:23752] 1任务
2016-04-17 11:09:47.029 GCDTest[624:23752] 2任务
2016-04-17 11:09:47.029 GCDTest[624:23782] <NSThread: 0x7fee6ae143a0>{number = 2, name = (null)}:2=0
2016-04-17 11:09:47.079 GCDTest[624:23782] <NSThread: 0x7fee6ae143a0>{number = 2, name = (null)}:2=1
2016-04-17 11:09:47.079 GCDTest[624:23782] <NSThread: 0x7fee6ae143a0>{number = 2, name = (null)}:2=2
2016-04-17 11:09:47.079 GCDTest[624:23782] <NSThread: 0x7fee6ae143a0>{number = 2, name = (null)}:2=3
2016-04-17 11:09:47.079 GCDTest[624:23782] <NSThread: 0x7fee6ae143a0>{number = 2, name = (null)}:2=4
2016-04-17 11:09:47.080 GCDTest[624:23782] <NSThread: 0x7fee6ae143a0>{number = 2, name = (null)}:2=5
2016-04-17 11:09:47.080 GCDTest[624:23782] <NSThread: 0x7fee6ae143a0>{number = 2, name = (null)}:2=6
2016-04-17 11:09:47.080 GCDTest[624:23782] <NSThread: 0x7fee6ae143a0>{number = 2, name = (null)}:2=7
2016-04-17 11:09:47.080 GCDTest[624:23782] <NSThread: 0x7fee6ae143a0>{number = 2, name = (null)}:2=8
2016-04-17 11:09:47.080 GCDTest[624:23782] <NSThread: 0x7fee6ae143a0>{number = 2, name = (null)}:2=9

```

图 11-4 Xcode 的打印信息

从打印信息分析可以发现，加入队列的两个循环任务是串行执行的，这与测试代码中创建的队列类型有关，第 1 次循环任务是在当前线程的主线程中执行的，第 2 次循环任务是在独立的子线程中执行的，这和开发者调用添加任务到调度队列所使用的方法有关。

GCD 的用法不只在管理线程执行任务，其中还有队列组、消息传递、队列挂起与开启等更多高级用法，这些不是本节重点，感兴趣的读者可以自行了解。



提示

在实际开发中，开发者常将耗时的任务放在子线程中执行，但有关界面 UI 刷新的操作一定要放在主线程中执行，否则界面的刷新会有延时。

本书特色

本书由经验丰富的iOS开发工程师编写，以iOS 9+Xcode 7+Objective-C为技术核心，通过大量的实战演练，将基础知识与开发实践相结合，系统地介绍了iOS从零基础开发到App Store上线的全部技术细节。通过阅读本书，读者能够完整地了解iOS应用开发的全流程，并学会如何开发一款兼容优雅的App产品。

本书教学实例

登录注册界面的搭建
手机网页浏览器的开发
“笑一笑”应用程序的开发
音频播放器的开发
Flappy Bird游戏的设计与开发

作者特别为本书录制了“iOS UI开发语音视频教程”，该视频教程详细地讲解了iOS UI设计的核心内容，包括基础篇、中级篇、高级篇、进阶篇、扩展篇5个部分，共36堂课，播放时长超过13小时，可以大幅提高读者快速深入学习iOS UI设计的效率。

为方便读者上机演练，本书还提供了iOS UI开发视频教程源代码和本书实例源代码，读者可以从本书提供的下载地址中获取。

“问啊” App联合创始人
李明偉

琿少以他扎实、专业的技术知识曾在“问啊” App上为很多人解答过问题，相信每位读者阅读这本书的每一分钟都会有回报！

千峰教育iOS教学部主管
吕志轩

大量的操作配图与详细的过程讲解是本书的一大特点，本书十分适合并无太多基础的读者学习与参考，相信通过阅读本书读者可以快速上手iOS应用程序的开发。



良师益友网责任编辑
罗 生

《iOS开发实战：从零基础到App Store上架》一书使我们眼前一亮，作者琿少从开发环境的搭建开始，一步步带领读者领略完整的iOS应用从开发到上线的整个过程。本书在章节中添加了大量的操作配图，十分易于读者操练与上手。在模块知识的讲解中，又穿插了与之相关的实战应用，如此深入浅出地将理论基础与实战结合的应用类书籍实属难得。无论你是即将毕业的大学生还是并无太多iOS开发经验的从业者，只要你想学习iOS开发，本书都将是你的不二之选。

麦子学院CEO
张凌华

琿少是麦子学院几百位合作讲师中最受上千名iOS企业直通班学员喜欢的老师之一。在《iOS开发实战：从零基础到App Store上架》一书中，琿少将iOS技艺从入门到精通都毫无保留、细致准确地写了出来，希望每一位iOS从业者会因为本书少走弯道，并快速奔上iOS大神之路。

清华大学出版社数字出版网站

WQBook  

www.wqbook.com

ISBN 978-7-302-44184-7



9 787302 441847 >

定价：69.00元